# MSX マシン語入門講座

湯浅 敬著




アスキー出版局

# MSX
# マシン語入門講座

湯浅 敬著

アスキー出版局

〔Book Cover〕
Illustrate by Shigenari Onishi
Design by Kenichi Samura／Number One Design Office

# はじめに

この本は，MSX ユーザーのためのマシン語入門書です．〝MSX を使っているけど，BASIC ではもの足りない〟，〝面白いゲームプログラムを作ってみたい〟という人に，ぜひ読んでもらいたいものです．

MSX はマシン語には向いていないといわれますが，決してそんなことはありません．確かにマシン語を扱うためにはそのためのソフトウェアが必要になりますが，実際に使ってみると，マシン語の細かいレベルでもプログラムが作りやすいように規格ができているのに驚かされます．

マシン語を勉強してみると，いままで BASIC だけを使っているときには知らなかったこと，たとえばコンピュータというのはこのようにして動いているとか，MSX にはこんな機能もあったということなどかわかるようになります．こうなれば，あなたの愛機 MSX の全貌が見えてきますし，認識も変わってくるでしょう．

本書は 17 章から成っています．まず 1 章はイントロダクションで，あまり本質的な話はでてきませんから気楽に読んでください．2 章では MSX という機械はどのようにできているか，3 章は MSX ではマシン語をどのようにして勉強するかという話です．この 2 つはマシン語を見ていく前には絶対に必要な知識ですから，しっかりと理解してください．

そしていよいよ 4 章から，マシン語の命令の解説を行います．マシン語にはさまざまな命令がありますが，それを 4 章から 14 章までに分けて解説していきます．この中では実際にその命令を使ってマシン語のプログラムを作って実行してみますから，MSX が手元で使える人は是非実際にプログラムを入力して実行してみてください（このためには 3 章で紹介する〝モニタ〟という BASIC プログラムが必要になります）．

その後 15 章からはマシン語の応用です．ここでは BASIC とのマシン語の結合，プログラムテクニック，それに画面の扱い方を説明していきます．

一通り最後まで読めば，マシン語の命令それに MSX でのマシン語の使い方がわかるでしょう．そうすれば MSX という機械がより親密なものに見えてくることと思います．

1984 年 盛夏　湯浅　敬

# 目　次

Illustrated by Kiwako Ishiwatari

# 1章
# マシン語ってどんなもの

# 1.1

# マシン語へのチャレンジ

この本では家庭用パーソナルコンピュータ〝MSX〞を使って，マシン語にチャレンジしていきます．多くのMSXのユーザー，特にまだコンピュータを始めたばかりの人にとっては，マシン語のイメージは〝とっつきにくい〞とか〝難しい〞というものになりがちですが，実際にはそんなに難しくないというのがこの本の主張です．ぜひ実際にマシン語に触れてみて，これまでの暗いイメージを吹きとばしてください．

まず最初に，この章ではマシン語のイメージについて述べます．これは〝マシン語とはいったい何か？〞，〝どうやって覚えるのか？〞というお話です．

マシン語を覚えるにはいろいろな予備知識が必要になります．それはコンピュータ内部のこととか，特殊な数の数え方とかいったものですが，多くの人にとってはまったく新しい知識です．BASICがわかるという人でも，いきなりこんなものを出されたのでは，〝え! 何なのこれ？〞と面くらってしまうでしょう．ですからその前に，準備体操のつもりでマシン語のイメージをある程度つかんでおこうというわけです．

## マシン語のイメージ

皆さんは，必ずどこかで〝マシン語〞という言葉に出会ったことがあると思います．どういうわけか，MSXに付いているユーザーズ・マニュアルにはこのマシン語(マニュアルには〝機械語〞と書いてあります)のことがあまり詳しく書かれていません．しかし，コンピュータ雑誌を読んだりマイコン仲間と話をしているうちに，自然に耳にはいってくるのがこの言葉なのです．

最初のうちはなんだかよくわからないものですが，マシン語でプログラムが作れるとカッコイイので，何となく〝憧れ〞の目で見てしまうものです．しかしこの言葉に出会うたびに，コンピュータを買ったばかり，BASICを始めたばかりという人もおそ

らく次にあげるようなマシン語のイメージができあがってくるでしょう.

- ○ プログラムを作るためのもの
- ○ 市販されているプログラム
- ○ スピードがある
- ○ 数字でできている
- ○ 作るのはたいへん

### ▶プログラムを作るためのもの

MSX には MSX-BASIC というものが付いており，この BASIC を使えばいろいろなプログラムが簡単に作れます．しかし，それでも世の中には〝マシン語で書かれたプログラム〟なるものが存在するのです．

マシン語も BASIC と同じくプログラムを作るためのものなのですが，BASIC とはいろいろな面で違っています．これは後で何度も出てきますが，BASIC とマシン語では，プログラムの作り方もできたプログラムもまったく違ったものになります．

### ▶市販されているプログラム




電器店やパソコンショップでテープや ROM カートリッジの形で売られているプログラムは，ほとんどがマシン語で作られています．MSX の場合はそのほとんどがゲームプログラムです．皆さんもきっとプレイしたことがあるでしょうが，あの有名な〝けっきょく南極大冒険〟や〝イリーガス〟といったゲームは，すべてマシン語で作られたプログラムです．

### ▶スピードがある

なぜマシン語のゲームがおもしろいかというと，マシン語で書いたプログラムはスピードがあるからなのです．たとえばあの有名な〝インベーダー〟というゲームがあ

ります．BASICでもこのインベーダーゲームのプログラムを作ることができます．しかし形だけはインベーダーゲームの格好をしていても，遅すぎてゲームとして〝遊ぶ〟ということはできません．

アクションゲームにとってスピードは命です．おもしろく楽しめるアクションゲームを作ろうというのであれば，BASICでは遅すぎてダメなのです．ですからこういうゲームプログラムは，どうしてもマシン語で作らなければなりません．

**▶数字でできている**

雑誌によく〝マシン語ダンプリスト〟なるものが載っています．それを見ると，最初から最後まで数字やアルファベットが並んでいます．〝何でこんなものがプログラムとして動くのか？〟それより〝何でこんなものを人間が作れるんだ？〟という気になってくるものです．でも世間にはこんなプログラムでも作ってしまうような，人間離れした人が結構いるのです．

**▶作るのはたいへん**

マシン語というのは数字でできているわけですから，プログラムを作るのはBASICよりもかなりたいへんなことに違いありません．なぜならBASICの場合，プログラム作り自体はたいへんな作業ですが，おのおのの命令自体は〝PRINT〟とか〝GOTO〟というようにやさしい英語でできています．それに対してマシン語の場合は，プログラミングそのもののめんどうくささに加えて，命令１つ１つが数字でできているわけですから非常にたいへんな作業になりそうです．

それからよく聞く話に〝暴走〟があります．これはマシン語のプログラムの中にミス(バグ)があると，まったく手の付けられない状態におちいってしまうというものです．マシン語のプログラマーは必ずこれに泣かされると聞きます．

BASICの場合プログラムの中にバグがあっても，その場所が簡単にわかってすぐに直すことができます．ところがマシン語の場合，プログラム作りのたいへんさに輪をかけて，デバッグ作業がたいへんなのです．たいていの場合，マシン語のバグの原因というのは非常につかみにくいものです．

# 1.2
# なぜマシン語を使うか?

MSX に限らず，パーソナルコンピュータは電源を入れた時点で BASIC が使えるようになっています．BASIC はプログラミング言語と呼ばれるように，これを使えばいろいろなプログラムを作ることができます．それなのに，なぜあえてマシン語などというよりメンドウなものを使うのでしょうか？

これはよく考えておかなければならない問題です．マシン語は BASIC に比べると何倍も，ものによっては何百倍(！)も速く実行される，だからといってプログラムはみんなマシン語で作った方がよい，というのは間違いです．

マシン語のプログラムを作るには，BASIC の何倍も時間がかかるということを忘れてはなりません．その点 BASIC には，プログラムを作りやすくするための様々な工夫がなされています．実行速度が遅いといっても，電卓でもできる程度の計算問題を実行させるのなら，わざわざマシン語でプログラムを作る必要はありません．

マシン語で作る価値のあるプログラムとは，何度も何度も繰り返し使うものです．また計算問題にしても計算量が膨大になってくれば，BASIC の処理速度の遅さというのも顕著になります．こんな場合もマシン語が求められるのです．

## BASIC の遅さ

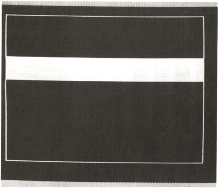
計算問題の答を求める以外に，プログラムの実行速度が必要になる場合があります．身近な例では，アクションゲームがこのマシン語の実行速度の威力をいかんなく発揮しているといえるでしょう．特にグラフィック画面を利用する場合，BASIC でも LINE 文とか CIRCLE 文を使って，いろいろな図形を描くことができますが，これらは BASIC の〝遅さ〟が目立つ部分なのです．

試しに**リスト 1.1** のプログラムを入力して実行してみてください．これは LINE 文

で枠を描き，PAINT 文でその中を塗るプログラムです．

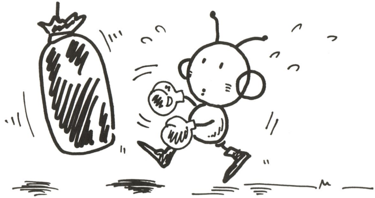
## リスト 1.1

```
100 SCREEN 2 ……………スクリーンモード2の設定(256×192)
110 LINE(10,10)-(200,170),8,B ……………枠を描く
120 PAINT (100,80),8 ……………枠の中を赤色で塗る
130 GOTO 130
```



PAINT 文は横の直線を何本も引いているのですが，注意して見ているとそれがわかります．ですから BASIC ではゲームの中でこういった図形を素早く(リアルタイムに)動かすようなことは，とうてい不可能なことなのです．こんなときにはどうしてもマシン語の助けが欲しくなります．

また，マシン語を使うと BASIC ではできなかったようなキメの細かい芸当ができます．これについては後で具体的な例を見てみましょう．



# 1.3

# BASICの動作

これまで，〝マシン語はプログラム作りがたいへんだ〟ということを述べてきました．それから〝マシン語は実行速度が速いが，BASICは遅い〟ということも述べました．しかし，これは具体的にはどういうことなのでしょうか？　いま使っているBASICというのは，プログラムを作るのがどう簡単で，なぜ遅いのでしょうか？

ここではマシン語と比較する意味で，〝どうしてBASICはプログラムが作りやすいのか〟，その代わり〝どうして実行速度が遅いのか〟という疑問に答を出してみたいと思います．

## BASICのしくみ

BASICはなぜ遅いのか？　それはBASICのしくみそのものに原因があります．この〝BASICのしくみ〟には，今後私達がマシン語にチャレンジしていく上で重要な鍵が潜んでいます．

さて，ここでまた1つプログラムを見てみましょう．

### リスト 1.2

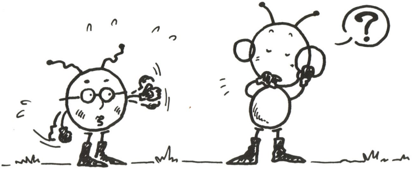
```
100 SCREEN 2  ……………スクリーンモード2の設定
110 FOR I=10 TO 120 STEP 10 ┐
120  CIRCLE(128,96),I,15    │(128,96)を中心とする同心円を12個描く
130 NEXT                    ┘
140 GOTO 140
```

**リスト1.2**は，同心円を12個描くプログラムです．このプログラムは，どのようにして画面に円を描くという作業を行うのでしょうか？

BASICプログラムはLIST命令を実行すればリストをとることができますし，RUN命令で何回でも実行することができます．これは，MSXがプログラムを覚えているからできることなのです．

コンピュータには**メモリ**というものがあり，プログラムを記憶することができます．BASICプログラムを入力すると，それはすべてこのメモリに記憶されていきます．プログラムのリストを表示したり実行したりするときは，メモリに記憶されたプログラムを見ているのです．

メモリに記憶されたプログラムを実行するのは**CPU**というものです．ところが，CPUはBASICプログラムを直接実行するというわけではありません．たとえば，ここに出ている〝SCREEN〃や〝CIRCLE〃といったBASICの言葉(命令)は，実はCPUには何のことだかサッパリわからないのです．CPUは，後で出てくるマシン語で書かれた命令しか理解できません．




では，このBASICプログラムはどうやって実行されるのでしょうか？ 実は，BASICプログラムとCPUの間を仲介しているものがあるのです．これはBASICのプログラムから1つずつ命令を読んでいきます．

たとえば**リスト1.2**の場合，最初にSCREEN文があります．これを読んだら，CPUに対して〝画面をグラフィックス用に設定する〃ということを実行させます．その次に〝FOR〃がありますから，ループがあることがわかります．そのループの中では，〝CIRCLE〃という命令で円を描きます．140行にくると，次に命令を読み出すところを140行にします．そこにはまた〝GOTO 140〃がありますから，ここで無限ループができるのです．

## BASIC インタープリタ

このように BASIC のプログラムはその命令１つ１つが CPU によって直接実行されているわけではなく，１つ１つの命令が，解釈されながら実行されます．こんなことをいったい誰がやるのかというと，**BASIC インタープリタ**というプログラムがやっているのです．MSX では，電源を入れたときから BASIC プログラムを入力して実行する間，つねにこのインタープリタが働いています．



どうしてこのようなめんどうなことが行われるのでしょうか？ それは，BASIC の命令というのが，FOR 文にしても PRINT 文にしても，人間にわかりやすい言葉を使うように作られているからなのです．CPU というのはコンピュータの中でのあらゆる仕事をやってくれるものなのですが，残念ながら人間ではなく機械なのです．機械には，BASIC プログラムの〝人間の言葉〟が理解できません．

そこでこの機械である CPU と，人間の言葉で書かれた BASIC プログラムの仲立ちをしてくれるものとして，BASIC インタープリタが存在するのです．このように機械との間にはいってくれるプログラムがあるため，前に述べたように，人間は楽にプログラムを作ることができるのです．

BASIC のプログラムが遅いのは，このインタープリタに原因があります．つまりプログラムに書いた命令を，まず読み出して，マシン語に通訳したものを CPU に実行させるということを行っているためです．

このように，BASIC というのは〝人間がプログラムを作りやすく〟するために，実行速度を犠牲にしているのです．

# 1.4
# マシン語の実行

さきほどは，BASICのプログラムはインタープリタという別のプログラムによって実行されているということを見たわけですが，ここではマシン語のプログラムは作成，実行の点でBASICとどのように違うかについて見てみましょう．

## マシン語は直接動く

マシン語のプログラムは，BASICと同じようにメモリに記憶されます．そしてCPUがそれを読み出して実行するわけですが，この過程がBASICとは大きく異なります．つまりマシン語はCPUが命令を〝直接〟読んで実行するのです．

BASICの場合には，1つ1つの命令がいわば〝人間の言葉〟で書かれていました．そのためCPUが直接実行することができず，間にインタープリタという〝通訳プログラム〟が存在していたのです．

ところがマシン語のプログラムというのはこのようなインタープリタをはさまずに，CPUが直接実行できるプログラムなのです．ですからCPUにいろいろな命令を与えるためには，人間の方がCPUに歩み寄って，CPUにわかる言葉，すなわち〝マシン語〟でプログラムを作ってやらなければならないのです．

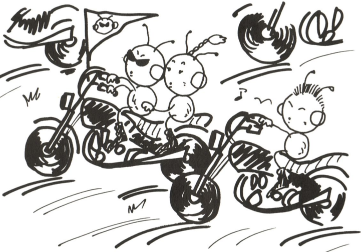
## 暴走が起こるわけ

BASICでは様々なエラー処理を行ってくれます．たとえばPRINT文を〝PRINY〟などと間違えて入力すると，インタープリタがその単語を読み出した時点で〝通訳できない〟ために，エラーメッセージを出してプログラムの実行は中断されます．この場合，CPUにはその命令を伝えないので命令が実行されずに終わるのです．

ところが，マシン語の場合はこうはいきません．プログラムの中に間違った命令があっても，CPU はそれを〝間違った命令〟としてそのまま実行してしまいます．ですからバグがあってもプログラムが止まらないのです．さらに危険なのは，この間違った命令によってプログラムが止まらなくなって手がつけられなくなったり，メモリの中のデータやプログラム自身をも消してしまうことがあるということです．これがいわゆる〝暴走〟です．

プログラムが暴走してしまった場合，どこでどうしてエラーが起こったのかを知るのはなかなか容易なことではありません．もちろんエラーメッセージというようなありがたいものはありません．またマシン語の場合，プログラムの実体はすべて数字ですから，その内容を読んでチェックするというのはたいへんめんどうな作業になります．

このようにマシン語はあくまで〝機械語(マシン)〟，プログラムを作る人間の方が CPU になったつもりで考えてやらなければプログラムは作れないということを，よく心にとどめておいてください．



## MSXとマシン語

MSX でマシン語のプログラムを作るなどというと，パパやママでも気軽に使えるホームコンピュータがマシン語とどんな関係があるのか不思議なくらいだ．でも本当は，MSX ぐらいマシン語のプログラム，特にゲームソフトを作るのに便利なコンピュータはほかにないといっていい．これは，MSX の備えている機能を考えてみるとよくわかるはずだ．まず，画面のモードがいくつもあるので目的に応じた画面が選べること，また，スプライト機能によってキャラクタが高速に操作できることがあげられる．ゲームにとって，ビジュアルな展開とスピード感は欠かせない要素だから，プログラム作りには非常に有利にできていることがわかるはずだ．

このほかにも，サウンド機能やジョイスティック，キャラクタが自由に設定できるなど数え上げればきりがないが，MSX でマシン語プログラムを作るときの本当のメリットは，こういった機能がマシン語で簡単に操作できるところにある．

実は，MSX にはあらかじめ上で述べたような機能をマシン語のプログラムとして用意している．これは，BIOS（バイオス：6章参照）と呼ばれるもので，これを使えばプログラムがやたら簡単になってしまうのだ．MSX のゲームソフトはかなりの数が売られているが，有用な BIOS が数多く用意されていることが，その理由となっているようだ．



# 2章

# MSXのハードウェアを調べる

# 2.1

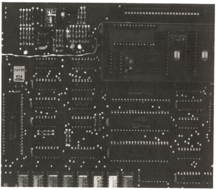
# MSXのしくみ

コンピュータを構成するものとして，**ハードウェア**と**ソフトウェア**というものがあります．ハードウェアというのは機械そのもののことで，主に回路のことを指します．ソフトウェアというのは，その機械の上で走らせるプログラムのことです．BASICのプログラムもソフトウェアの1つです．

BASICを使っている間は，ソフトウェアのことだけ考えていてもたいていのことはできたはずです．しかしマシン語を使うためには，どうしてもハードウェアのことを知らなくてはなりません．どうしてでしょうか？ それはマシン語でのプログラミングというのは，CPUを中心としたコンピュータの機械(ハードウェア)を動かすものだからです．

そこで，ここではMSXのハードウェアについて解説していきます．これは，マシン語を始める前にどうしても必要な基礎知識ですから，しっかりと理解してください．説明の中でBASICでも確かめられることは，そのためのプログラムを載せておきますので，MSXを持っている人はぜひ実際にプログラムを入力してみて，身をもって確かめて欲しいと思います．

## MSX マシンの中



まず最初にMSXの中身，ハードウェアの大まかな構造について見てみます．手っとり早くMSXの中身を見たければ，機械を開けてみるのがいちばんです．実際に開けてみると，右の写真のような複雑な回路が現れます*．

＊故障の原因になることが多いので，読者の皆さんはMSXを開いてはいけません．

四角くて黒いものが，ICとかLSIと呼ばれるもので，その中にはさらに複雑な回路が詰まっています．これらはプリント基板(回路の配線がプリントされているところからこう呼ばれている)という板にささっていて，その間が電線で結ばれて1つの巨大な回路になっているのです．

いきなりこんなものを見せられて，驚いた人もいるでしょう．でも安心してください．いくら何でもこんなものを理解しようというのではありません．コンピュータの回路というのは，非常に大きく複雑にできています．小学校の理科でやったように，豆電球と電池の配線をたどって回路の働きを知るというわけにはいきません．このようなことは，実際問題として不可能です．

そこでコンピュータの働きを覚えていくためには，**図2.1**のように主要な部分だけを取り出して，その部分の働きを知るというのがふつうです．この図からは，CPUとメモリがつながっており，メモリにはプログラムとデータがはいることがわかります．

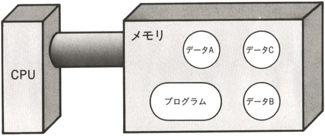


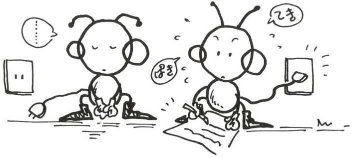

**図2.1 CPUとメモリ**

## メモリの構造……RAMとROM

1章でも述べましたが，コンピュータの中心的な働きをしているところといえば，CPU，それにメモリです．特にCPUというのはコンピュータの中で最も大切なところで，あらゆる仕事の中心的役割を果たします．コンピュータの中の他の回路は，このCPUの指令によって〝動かされている〟といってもよいほどです．

それに対してメモリというのは，プログラムを実行したり計算したりする能力はありませんが，コンピュータが様々な処理をする上で必要となるありとあらゆる情報(プログラムやデータ)を覚えているところです．

メモリにはROM(Read Only Memory)と呼ばれるものと，RAM(Random Access Memory)と呼ばれるものの2つの種類があります．このうち，ROMはコンピュータの電源を切っても中の情報は消えませんが，その記憶されている情報を書き換えることはできません．それに対してRAMは自由に中身を書き換えることができますが，電源を切るとその情報はすべて消えてしまいます．




もちろん，MSXのメモリにもROMの部分とRAMの部分があります．BASICの命令にメモリの内容を調べたり書いたりするPEEK関数，POKE文というものがありますから，実際にROMとRAMがどんなものかを見てみましょう．

まずメモリの内容を調べるのがPEEK関数です．次のように入力して結果を確かめてください．

```
                ┌─メモリの60000の場所を指定する
PRINT PEEK(60000)⏎ ……………リターンキーを押す
 0 ……………メモリの60000の場所の内容は0
Ok
█
                              ＊下線部分は入力した部分を指す
```

PEEKの後ろのカッコの中にはメモリの〝どこの場所か〟という数字を書きます．つまり〝0〟という結果の出力は，〝メモリの60000の場所には0という数字がはいっている〟ということを意味します．

次に POKE 文を使ってこの内容を書き換えてみます．

```
POKE 60000,100⏎ ……………メモリの60000の場所に100を書き込む
Ok
■
```

上の POKE 文は，メモリの 60000 という場所に 100 という数を書き込む命令です．果たしてうまくいったでしょうか？ もう一度 PEEK 関数で 60000 の場所を見てみましょう．

```
PRINT PEEK(60000)⏎ ……………メモリの60000の場所の内容を表示する
 100 ……………前に書き込んだ100が確認できる
Ok
■
```

うまくいきましたね．これでメモリの内容を書き換えることができたのです．このように値を書き換えることができるということは，60000 という場所は RAM であるということです．

ところでさっきからメモリには数字しかはいっていないようですね．試しに〝A″という文字を入れてみましょう．

```
POKE 60000,"A"⏎ ……………メモリ60000の場所に〝A″という文字を書き込んでみる
Type mismatch ……………タイプミスマッチエラー（数値関数に文字列を使ったために
                          データ型が不一致のエラーが出た）
Ok
■
```

〝Type mismatch″というエラーが出てしまいました．ということはうまく POKE 文が実行できなかったということです．このように，メモリには数字しか入れることができません．このことは重要ですからよく覚えておいてください．

さて，今度は別の場所をやってみましょう．0という場所のメモリの値を調べて，書き換えてみます．

```
PRINT PEEK(0)⏎ ……………メモリの0の場所の内容を表示する
 243 ……………メモリの0の場所の内容
Ok
POKE 0,100⏎ ……………メモリの0の場所に100を書き込む
Ok
■
```

最初0の場所には243という数字がはいっていたわけですが，これをまた100に書き換えてみました．〝Ok〞と出ているので書き換えに成功した模様です．再度PEEK関数で確かめてみましょう．

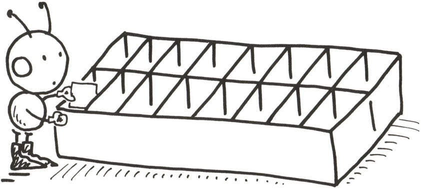
```
PRINT PEEK(0)⏎ ……………もう一度メモリの0の場所を表示してみる
 243 ……………書き換えられていない(メモリの0の場所はROMであるためにPOKE文では
Ok               書き換えることはできない)
■
```

おや，変わっていませんね．書き換えができませんでした．これは，この0という場所はROMだということを示しています．ここには電源を入れる前から243がはいっていて，書き換えることができないのです．POKE文を実行したときには実際に100を入れたつもりなので〝Ok〞と出たのですが，ROMのメモリは実際には書き換わりません．

このようにメモリの中にはROMとRAMがあって，RAMの方は(いとも簡単に)内容を書き換えることができますが，ROMの方は絶対に内容を書き換えることができません．念のため一度電源を切って，再度電源を入れた後，いまの2つの場所を見てください．0の方は相変わらず243ですが，60000の方にはいま入れた100は残っていないことがわかるでしょう．

## CPU とメモリの関係

**図2.1** で見たように，メモリはCPUとつながっています．CPUが何か仕事を行う場合は，メモリからプログラムを読み出す必要があるからです．プログラムを実行するときには，いろいろなデータが必要になったり計算結果が出たりします．こういうデータも，すべてメモリの中に置いておきます．ですからメモリの中にはプログラムもデータもはいるのです．そしてCPUはこの両方を扱うため，メモリを読んだり書いたりすることができます．



〝データ〞といっても何か漠然としていますが，これはプログラムを実行する上で使うあらゆる値のことを指しています．たとえばBASICの変数のようなものはデータです．いまXという変数を使っているとします．CPUはXという変数の値が必要なときはメモリ中のXの値が記憶されている場所から値を読み出してきますし，Xに代入するときにはその場所の内容を書き換えるのです．

CPUというのは計算はできますが，たくさんのことを覚えることができません．ところがこのようにメモリを使っているために，たくさんの変数や配列を使ったプログラムでも実行できるのです．

# 2.2

# メモリとプログラムの記憶

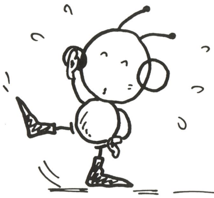
前の節ではプログラムがメモリの中に記憶されるということを述べました．そこでここでは実際に私達が作るBASICプログラムが，どのようにメモリの中にはいっているかを調べてみます．

BASICのプログラムは，RUNという命令で何度でも実行できます．これは，そのプログラムがメモリに記憶されているからできることなのです．しかしキーボードから打ち込んだプログラムがメモリに記憶されるためには,次のような問題があります．

**① メモリには数字でしか記憶されない(前の節でPOKE文で見たとおり)**

**② CPUは本来マシン語で書かれたプログラムでなければ実行できない**

つまり，まずどうやって文字の集まりであるBASICのプログラムを数字しかはいらないメモリに記憶させているかということ，そしてそれをどうやってCPUが実行しているかということです．



まず①の問題ですが，BASICのプログラムの中に現れる文字は，すべて数字に変換されて記憶されます．BASICにCHR$とASCという関数があるのを知っていますか？ MSXの中では，**表2.1**のキャラクタコード表に示されるように，1つの文字に0から255までの数を割り当てています．

| キャラクタコード | 文字 | キャラクタコード | 文字 | キャラクタコード | 文字 | キャラクタコード | 文字 |
|---|---|---|---|---|---|---|---|
| 0 | | 64 | @ | 150 | か | 214 | ヨ |
| 1 | | 65 | A | 151 | き | 215 | ラ |
| 2 | 制 | 66 | B | 152 | く | 216 | リ |
| 3 | 御 | 67 | C | 153 | け | 217 | ル |
| 4 | 用 | 68 | D | 154 | こ | 218 | レ |
| 5 | コ | 69 | E | 155 | さ | 219 | ロ |
| 6 | ー | 70 | F | 156 | し | 220 | ワ |
| 7 | ド | 71 | G | 157 | す | 221 | ン |
| 8 | | 72 | H | 158 | せ | 222 | ゛ |
| 9 | | | I | 159 | そ | 223 | ゜ |
| 34 | ″ | 98 | b | 160 | | 224 | |
| 35 | # | 99 | c | 184 | | | |
| 36 | $ | 100 | d | 185 | ク | 249 | る |
| 37 | % | 101 | e | 186 | コ | 250 | れ |
| 38 | & | 102 | f | 187 | サ | 251 | ろ |
| 39 | ’ | 103 | g | 188 | シ | 252 | わ |
| 40 | ( | 104 | h | 189 | ス | 253 | ん |
| 41 | ) | 105 | i | 190 | セ | 254 | |
| 42 | * | 106 | j | 191 | ソ | 255 | |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | | |

**表2.1 キャラクタコード表の一部**

このキャラクタコードと実際の文字との対応を求める関数がCHR$とASCなのです．これらは次のようにして使われます．まずASC関数は引数に文字をとります．次に示す例では〝A″のキャラクタコードは，65であることを示しています．逆にCHR$を使うと，ある番号の文字を出すということができます．

```
PRINT ASC("A")↵
 65 ……………〝A″という文字のキャラクタコード
Ok
PRINT CHR$(67)↵
C ……………キャラクタコードが67の文字
Ok
■
```

コンピュータにとって，文字に数字が対応しているというのは重要なことです．つまりメモリの中には数字という形でしか記憶できなくても，それが文字と対応していれば文字を数字の形で扱うことができるのです．BASICのプログラムも，メモリの中では文字が数字に変換された形で記憶されています．

## プログラムが記憶されている様子

ここでBASICプログラムがメモリにどのように記憶されているかを実際に確かめてみましょう．リスト2.1の〝DUMPME〟というプログラムを入力してください．これはBASICのプログラム(ここではDUMPME自身)がメモリの中でどのように記憶されているかを画面に表示するプログラムです．プログラムの実行には関係ありませんが，いちおうREM文の行もそのまま入力してください．

### リスト 2.1

```
100 REM DUMP ME
110 SCREEN 0
120 S=PEEK(&HF676)+PEEK(&HF677)*256
130 E=PEEK(&HF6C2)+PEEK(&HF6C3)*256-2
140 FOR A=S TO E STEP 8
150  FOR J=0 TO 7
160   D$=RIGHT$("0"+HEX$(PEEK(A+J)),2)
170   PRINT D$;" ";
180  NEXT
190  PRINT "  ";
200  FOR J=0 TO 7
210   X=PEEK(A+J)
220   IF X<32 THEN X=46
230   PRINT CHR$(X);
240  NEXT
250  PRINT
260 NEXT
270 END
```

このプログラムを実行すると図2.2のように画面に表示されます．左に8つずつ並んでいる数(16進数になっている)が実際にメモリにはいっている数で，その右に出ているものがそれぞれの数が意味する文字です．この結果を見るとREM文に書いた〝DUMP　ME〟という文字などはそのままメモリにはいっているのがわかります．

ところが，プログラムに書かれているPOKEとかPRINTという文字が見当たりません．実はこれらBASICの予約語は〝中間コード*〟というものに変換されているので，数字を見ただけではこれがどの命令が変換されたものかはわかりません．

---

* BASICの予約語を，短縮したコードに置き換えたもの．

ほかにも変数名やダブルクォーテーション(〝 〞)でくくった文字などはそのまま記憶されているようですね．興味があれば，プログラムの終わり(ENDよりも後ろ)にいろいろ文を入れて確かめてみるのもよいでしょう．

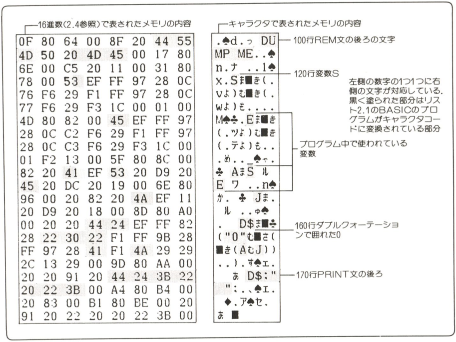



図2.2 DUMPME実行例

さてRUN命令を実行すると，メモリに記憶されたBASICプログラムが〝実行〞されます．しかし図2.2のような形のプログラムを，そのままCPUが実行するわけではありません．なぜならこれはBASICのプログラムを数字の形に変換しただけですから，あくまで〝人間の言葉〞で〝マシンの言葉〞ではないからです．

そこで，人間の言葉で書かれたBASICプログラムをCPUが実行するためのものとして登場するのが，1章にも出てきたBASICインタープリタなのです．

メモリの中には〝FOR〞とか〝PRINT〞というようなBASICの命令が，中間コードの形ではいっていますから，それらを順番に読み出して〝FOR〞なら繰返しのための処理を，〝PRINT〞なら画面に文字を表示する処理を行うのです．

# 2.3

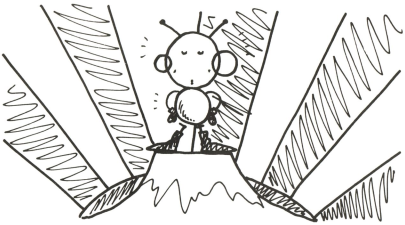
# CPU，メモリ，I／O，バス

ここまでのところで，メモリの様子についていくつかわかったことがあります．まずメモリには数字という形ですべてが記憶され，プログラム(BASIC プログラムもインタープリタのようなマシン語のプログラムも)とデータがここに記憶されていました．

ここでは，このようにメモリの中に記憶されたプログラムやデータを，CPU がどのように扱うのかということを見ていきましょう．またこの中で，外部とデータをやりとりするための**I／O ポート**，それに**バス**というものについて説明していきます．

## CPU の働き

CPU はコンピュータの中で，中心的な役割を果たします．その主な仕事は計算であるといいましたが，その働きをもう少し具体的に見てみましょう．



たとえば，BASICで〝X＝X＋3〟という命令を実行するとしましょう．変数Xの値はメモリの中に記憶されています．しかしメモリには，計算能力がありません．そこでこの式を実行するために，この変数Xの値を一度CPUの中に持ってきて，それに3を加えるという演算を行い，その後，再びもとのメモリの場所にその値を入れるのです．このように，実際に計算が行われるのはCPUの中だけなのです．

またCPUは，他の回路のコントロールも行います．これは音を出すPSGや，画面関係の処理を行うVDPといったLSIを動かしたりするのです．これについては後で〝I／Oポート〟のところで説明します．

## メモリ

メモリは大量のデータやプログラムを，一体どのようにして覚えているのでしょうか？ それを知るためには，まずメモリの構造を理解しなければなりません．

メモリは**図2.3**のように細かい部屋に区切られていて，その1つ1つが数字を覚えるようになっています．そしてそれぞれの部屋には，他の部屋と区別するため**アドレス**(日本語でいうなら**番地**)と呼ばれる番号が付いています．MSXの場合メモリの部屋は数万個ありますから，アドレスもその分だけ指定できます．

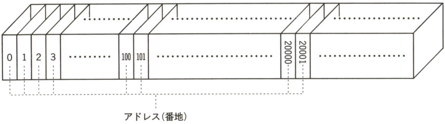



**図2.3 メモリの構造**

このアドレスというのはCPUがメモリとの間でプログラムやデータを読み書きするのに使われます．たとえば**図2.4**のように，メモリの内容を知りたいときには，〝何番地のメモリの内容は何だ？〟とメモリに尋ねるのです．逆にメモリに何か書きたいときには，〝何番地のメモリに何という数を書け〟とメモリに命じます．

この2つがCPUがメモリに対して行う操作のすべてです．前者を**読み出し**(READ)といい，後者を**書き込み**(WRITE)といいます．そしてこの2つを合せて〝メモリにアクセスする〟といったりします．RAMの〝Access〟というのがこの意味です．

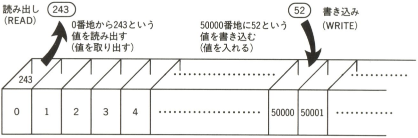



**図2.4 メモリアクセス**

BASICの命令でいうと，PEEK関数が読み出し，POKE文が書き込みを行います．**2.1 MSXのしくみ**では，どちらの場合も0や60000といった数字は〝メモリの場所(PEEK関数のカッコの中やPOKE文の最初のパラメータ)を指定する〟といいましたが，実はこれがこのアドレスなのです．

## バスの働き

CPUがメモリをアクセスするためには，CPUとメモリを結ぶものがなければなりません．それがここで説明するバスというものです．バスは**図2.5**にあるように，CPUとメモリをつないでいます．

バスは，データが通る道のようなものです．コンピュータの中の回路はこのバスを通して，他の回路とデータのやり取りをします．実際にはバスにある数値を表す電圧パターンを出力します．受け取る側ではバスからの電圧パターンによって，どのような数値が送られてきたかがわかるのです．

CPUとメモリの間には，**アドレスバス**と**データバス**という2種類のバスがあります．アドレスバスは16本の線で，データバスは8本の線でできています．CPUがメモリに対してアドレスを指定するときはアドレスバスが使われ，データを読み書きする際にはデータバスが使用されるのです．

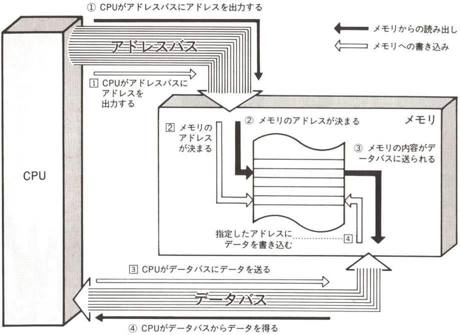



図2.5 バスの働き

CPUがメモリの内容を読み出す場合，CPUは読み出したいメモリのアドレスをアドレスバスに出力します．するとメモリの方では，どのアドレスが指定されたかということがわかりますから，その内容をデータバスに出力します．そこでCPUがデータバスを読み出せば，そのアドレスに何が書いてあったかがわかるのです．

メモリに値を書き込みたいときは，CPUはアドレスバスとデータバスに，それぞれアドレスとそこに書き込みたいデータを出力します．するとメモリの方ではアドレスとデータがわかりますから，メモリのそのアドレスにデータを書き込んでくれます．

ちなみに電圧パターンというのは，バス1本にかかる電圧が高いか低いかということです．ですからバス1本では，0または1という2通りの数字を表せます．これが2本になれば$2^2(2\times2)$で4通り，3本になれば$2^3(2\times2\times2)$で8通りの数字が表せるのです．

データバスは8本ですから0から255まで256通り($2^8$)，アドレスバス16本で0から65535まで65536通り($2^{16}$)の数を表すことができます．

## I/Oポート

さて，コンピュータの中で行われていることというのは，本質的にはこれまでに述べたことがほとんどです．つまりCPUがメモリに記憶されたプログラムを実行し，その中ではメモリにあるデータをアクセスするということを繰り返しているのです．しかし，これだけでは人間がコンピュータをコンピュータとして使うことができません．なぜならメモリの中にいくら計算結果を書き込んでも，それがCPUからしか読むことができないのでは何をやらせるかを伝えることも，やらせた結果を知ることもできないからです．

つまり，コンピュータの中から外部に対してデータを出してやることができなければなりませんし，また逆に，外からコンピュータに対してデータを入れてやることができなければなりません．ですから，コンピュータには外部との交信のための機能が付いています．これを**I/Oポート**といいます．

I/Oポートは図2.6に示すようにCPUとつながっています．MSXの場合I/Oポートは全部で256個あり，メモリのアドレスのようにポート番号が付いています．

I/Oポートのポート番号の指定にはアドレスバスのうち8本を使い，これで256通りの数を指定します．またデータバスはメモリのバスを共有しています．

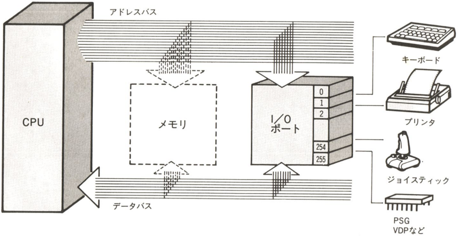



図2.6 I/OポートとCPUとの交信

# 2.4

# 2進数と16進数

前節までにMSXのハードウェアに関してはある程度のことを説明しましたが，もう1つマシン語をやる上でどうしても必要なものがあります．それがここで説明する2進数と16進数というものなのです．これらは数の数え方の1つなのですが，特にコンピュータのハードウェアとか，マシン語とは密接な関係があるものです．

それではまず，私達が日頃使っている数の数え方について見てみます．ふだん生活の中で数を数えているときには特に意識しませんが，私達の数の数え方は10進法というものです．これは1，2，3と数を数えたときに，8，9の次に1つ桁が上がって10になるもので，1つの桁にはいる数字は0から9までの10種類です．

数を数字で表したとき，その桁には重要な意味があります．たとえば424という数字があったとき，最初の4と最後の4では意味が違います．最初の方は100の位ですから，この4は〝400〞の意味，3番目の4は1の位ですからただの〝4〞です．同じように10進法で数が表現されていると，その意味するものは次のようになります．

1234＝1＊1000＋2＊100＋3＊10＋4

〝なんだ，こんなことあたりまえじゃないか〞といわれるかもしれません．そうです，確かにこのことは小学校で習うような当たり前のことです．というより10進法を使っている限り，上のような式は考えない方が普通です．

ではなぜこんな話を始めたかというと，数の数え方は10進法だけではない，ということをいいたかったからなのです．これから説明する2進法と16進法がそうなのです．

## 2進数

まず2進法の数というのは，各桁にはいる数字が0と1しかないというものです．

たとえば2進法で数を数えると，0の次に1，その次にはもう1の位に入れる数字がないので，桁が上がって10になります．次が11で，その次はまた桁上がりして100になります．以下10進法と比較してみると表2.2のようになります．

| 10進数 | 2進数 | 10進数 | 2進数 |
|---|---|---|---|
| 0 | 00000000 | 11 | 00001011 |
| 1 | 00000001 | 12 | 00001100 |
| 2 | 00000010 | 13 | 00001101 |
| 3 | 00000011 | 14 | 00001110 |
| 4 | 00000100 | 15 | 00001111 |
| 5 | 00000101 | 16 | 00010000 |
| 6 | 00000110 | 17 | 00010001 |
| 7 | 00000111 | 18 | 00010010 |
| 8 | 00001000 | 19 | 00010011 |
| 9 | 00001001 | 20 | 00010100 |
| 10 | 00001010 | ⋮ | ⋮ |

**表2.2　2進数と10進数**

MSX-BASICには2進数を扱う機能が付いています．&BとBIN$がそうです．まず&Bですが，これは2進数で表現した〝数″が10進数ではいくつになるかを表します．たとえば〝101100″を10進数に変換したければその前に&Bを付けるだけでよいのです．

```
PRINT &B101100⏎
 44 ……………2進数〝101100″を10進数に変換した値
Ok
■
```

BIN$はこの逆で，カッコでくくった数を2進数表記に直します．たとえば次の例を見てください．

```
PRINT BIN$(44)⏎
101100 ……………10進数の〝44″を2進数に変換した値
Ok
■
```

これは，44を2進数にすると101100になるということです．ほかにもいろいろな数を試してみてください．

## ビット／バイト

2進数で表した数の1つの桁のことを〝1ビット(bit)〟といいます．ビットというのは情報量の単位で，〝0か1か〟，〝ふたつにひとつ〟という意味です．この情報量というのは，〝どれだけの数が表せるか〟という意味です．たとえば1ビットならば0と1という2つの数が表せますし，2ビットになると0，1，10，11という4つの数が表せます．一般にNビットあれば，$2^N$個の数が表せます．

では，なぜコンピュータでは2進法を使うのでしょうか？ それはコンピュータの内部で数をどういう形で表しているかというところに原因があります．前にも触れましたが，コンピュータの中の情報(アドレスやデータ)はすべてが〝電圧が高いか低いか〟という電圧パターンで表されています．ですからこれを1つのビット，つまり2進数の1桁で表現するわけです．

たとえばデータバスの場合，**図2.7**に示すように8本の線がそれぞれ電圧が高いか低いかで1ビットを表しているわけですから$2^8$で256通りの数が表せます．最初が1でなく0ですから，0から255までの数が表せます．つまり8ビットのデータ1つ1つは0から255までの数なのです．またアドレスバスは16ビットですから0から65535($=2^{16}-1$)までの数を表すことができます．

コンピュータの中ではデータバスのように，よく8ビット単位でデータを扱います．8ビットのことを**1バイト**(byte)と呼び，1バイトでは0から255まで，256通りの数を表すことができます．

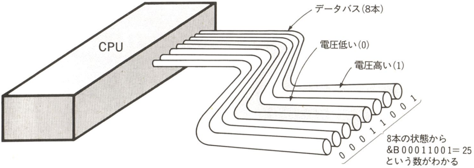



**図2.7 バスと2進数**

## 16 進数

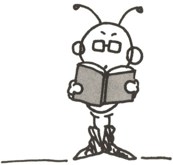
さてもう 1 つ 16 進法というのも必要になります．これは文字どおり 1 つの桁に 16 種類の数字を入れることができます．といっても私達が使っている数字は 10 進数ですから，〝数字〟と呼べるものは 10 種類しかありません．そこで 16 進数では足りない 6 つの数をアルファベットで補っています．

16 進法で数を数えると，0，1，2……ときて，8，9，その次が 10 ではなくてAになります．そしてその後B，C，D，E，Fときて，その後桁が上がって１０になるわけです．その後１９まできたら，１A，１B…１Fで，その次は２０になります．**表 2.3** に 10 進数，２進数，16 進数の対応を示しておきますので，よく見比べてください．

| 10進数 | 2進数 | | | | 16進数 | 10進数 | 2進数 | | | | 16進数 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0000 | 0000 | 0000 | 0000 | 0000 | 11 | 0000 | 0000 | 0000 | 1011 | 000B |
| 1 | 0000 | 0000 | 0000 | 0001 | 0001 | 12 | 0000 | 0000 | 0000 | 1100 | 000C |
| 2 | 0000 | 0000 | 0000 | 0010 | 0002 | 13 | 0000 | 0000 | 0000 | 1101 | 000D |
| 3 | 0000 | 0000 | 0000 | 0011 | 0003 | 14 | 0000 | 0000 | 0000 | 1110 | 000E |
| 4 | 0000 | 0000 | 0000 | 0100 | 0004 | 15 | 0000 | 0000 | 0000 | 1111 | 000F |
| 5 | 0000 | 0000 | 0000 | 0101 | 0005 | 16 | 0000 | 0000 | 0001 | 0000 | 0010 |
| 6 | 0000 | 0000 | 0000 | 0110 | 0006 | 17 | 0000 | 0000 | 0001 | 0001 | 0011 |
| 7 | 0000 | 0000 | 0000 | 0111 | 0007 | 18 | 0000 | 0000 | 0001 | 0010 | 0012 |
| 8 | 0000 | 0000 | 0000 | 1000 | 0008 | 19 | 0000 | 0000 | 0001 | 0011 | 0013 |
| 9 | 0000 | 0000 | 0000 | 1001 | 0009 | 20 | 0000 | 0000 | 0001 | 0100 | 0014 |
| 10 | 0000 | 0000 | 0000 | 1010 | 000A | 21 | 0000 | 0000 | 0001 | 0101 | 0015 |

**表2.3　2 進数，10進数，16進数**

この 16 進数というのは 2 進数で表現された数を短縮するのに使います．上の対応表をよく見ると，2 進数で表した 4 桁が 16 進数で 1 桁に表現できているのがわかります．つまり，1 バイト（8 ビット）の数は 16 進数では 0 からＦＦまでの 2 桁の数で表すことができます．このように，2 進数で表すと桁が長くなりすぎて間違えやすくなってしまいますので，16 進数を使うわけです．



ところでMSX-BASICには16進数を扱うための機能も付いています．HEX$と&Hがそうです．使い方はBIN$，&Bとまったく同様ですから試してみてください．

リスト2.2は10進数で入れた数を2進数や16進数に変換するプログラムです．またリスト2.3は2進数を16進数に変換する問題形式のプログラムです．このように2進数を16進数に変換するということが後で必要になりますから，このプログラムを使っていまのうちからこの操作に慣れておきましょう．

## リスト 2.2

```
100 'CONVERT 10 -> 2,16
110 PRINT "10ｼﾝｽｳ ｦ ﾆｭｳﾘｮｸ ｼﾃｸﾀﾞｻｲ"
120 INPUT X
130 B$=RIGHT$(STRING$(15,"0")+BIN$(X),1
6)
140 H$=RIGHT$("000"+HEX$(X),4)
150 PRINT " 2ｼﾝｽｳ ﾊ ";B$
160 PRINT "16ｼﾝｽｳ ﾊ ";H$
```

## リスト 2.3

```
100 'EXERCISE BINARY TO HEXADECIMAL
110 R=RND(-TIME)
120 X=INT(RND(1)*65536!-&H7FFF)
130 B$=RIGHT$(STRING$(15,"0")+BIN$(X),1
6)
140 FOR I=0 TO 3
150  PRINT MID$(B$,I*4+1,4);" ";
160 NEXT
170 PRINT
180 INPUT "16しん では いくつ?";H$
190 A=VAL("&H"+H$)
200 IF A<>X THEN PRINT"まちがい":BEEP:GOTO
 180
210 PRINT "せいかい!!"
220 GOTO 120
```

# 2.5
# CPUの内部

さていよいよCPUの内部を見てみましょう．これがわかれば，MSXというコンピュータのいちおうの動き方というものがわかります．ですから，ここを読めば晴れてマシン語に入門できるのです．

CPUの中で行われる仕事というのは，ほとんどが〝計算〟です．この計算の中には足し算，引き算のほかに，後で説明する論理演算（8章参照）というものがあります．BASICのLINE文やCIRCLE文といったグラフィック命令の実行も，実は単純な計算の組合せによって行われています．ですからCPUの中というのはいろいろな計算をするための回路が詰まっているのです．

## CPUの中の回路

MSXのCPUはZ80という名前で呼ばれているものが使われています．このZ80というCPUは，現在パーソナルコンピュータで使われているCPUの中では最もポピュラーなものです．MSXでマシン語のプログラムを作るには，まずこのZ80のことを知っておく必要があります．

CPUの中は図2.8のようになっています．この図にあるように，CPUの中は**レジスタ**というものがいくつかありますが，これは1バイトないしは2バイトのデータを記憶しておくものです．

レジスタの数というのは限られていますので，メモリのようにたくさんのデータを記憶することはできませんが，計算ができるというのが大きな特徴です．ふつうメモリにあるデータに対して足し算や引き算を行うときは，それをいったんCPUの中のレジスタに持ってきて演算を行います．

図2.8の中にあるALUというのが**演算回路**といって，実際にCPUの中で各種演算

が行われるところです．しかし後でも述べるように，プログラムを作るときにはこのALUのことはまったく意識する必要はありません．

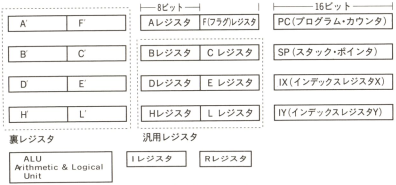



図2.8 CPUの内部

## メインレジスタ

レジスタは種類によってその機能や役割が異なります．まず**アキュムレータ**と呼ばれるAレジスタ，それに**汎用レジスタ**と呼ばれるBからLまで，この7つがよく使われるレジスタです．これらが扱う値の大きさは8ビットですから，レジスタ1つで0から255(2進数で11111111)までの数を持つことができます．

演算は通常Aレジスタで行われます．たとえばAレジスタの内容にBレジスタの内容を足すことができます．これは実際には，**図2.9**のようにAレジスタの内容とBレジスタの内容がALUにはいり，その和がAレジスタに入れられるという形で行われます．しかしALUの存在は特に意識する必要はありません．ただ〝Aレジスタ(の内容)にBレジスタ(の内容)を足す〞と考えればよいでしょう．

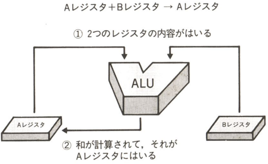



図2.9 ALUの働き

Aレジスタ以外の6つの汎用レジスタは直接その上で演算をすることはできません．つまりBレジスタにCレジスタを足したり，HレジスタからLレジスタを引いたりということはできないのです．しかし，これらは2つずつペアになって16ビットのデータを扱うことができます．たとえば〝BCレジスタ〟というのはBレジスタとCレジスタの内容をくっつけて16ビットにしたものです．この場合その数は次のようになります．

(Bレジスタの内容)＊256＋(Cレジスタの内容)

**Fレジスタ**というのはフラグレジスタといって，演算の結果起こる様々な状態を入れるためのレジスタです．これも8ビットですが，汎用レジスタのように使うことはできません．

汎用レジスタとA, Fレジスタには**裏レジスタ**というものがあります．これは図2.8にA'とあるように，〝'〟が付いたものです．これらは表と同じ働きをします．裏と表を取り換えるということはできますが，裏と表を同時に使うことはできません．

## その他のレジスタ

さて，CPUの中にはアキュムレータや汎用レジスタのほかにもいくつかのレジスタがあります．

まずPCはプログラム・カウンタといって，CPUがプログラムを読み出すときに，メモリのアドレスを指定するのに使われます．CPUはPCの指すメモリからプログラムを読み出し，実行するのですが，その命令を実行する前にPCは次の命令のあるところまで自動的に進められます．

SPはスタック・ポインタというもので，これもプログラムの中でメモリのアドレスを覚えておくのに使われます．ただこの働きは少し複雑なので，7章で説明します．

IXとIYはインデックスレジスタといってアドレスを指定するものですが，これも5章で実際に使いながら説明していきます．

最後にR, Iという2つのレジスタが残りましたが，これについてはこの本では使用しないので説明しません．これは〝割り込み〟というものを使うときに使用するものですが，本書では扱いません．

# 2.6

# MSXのメモリ構成

ここまでの話でMSXの構造，とりわけCPUとメモリのしくみやその構造についてはだいぶわかったことと思います．そこで実際に私達がメモリを扱う上で，どこのメモリが使えるのかということを見てみます．

## メモリのアドレス

図2.10はメモリの構造を示したものです．メモリの各アドレスには，8ビットのデータを入れることができます．またアドレスの指定は16ビットで行われます．16ビットの数を16進数で表示すると，0H番地からFFFFH*番地までで，これがアクセスできるメモリの領域です．

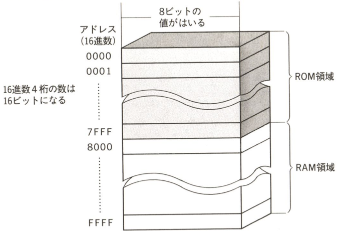



図2.10 メモリの構造

＊数字の後のHはHEXADECIMAL（ヘキサデシマル）のHで，その数が16進数であることを示します．

CPUの中の16ビットのレジスタ，すなわちPCやSP，またHとLを組み合せたHLなどはたいていこのメモリのアドレスを指定するために使われます.

## メモリアドレスの数値

メモリのＦＦＦＦH番地というのは，10進数で書くと65535番地になります．しかし，今後マシン語を使っていく場合には，メモリのアドレス指定には16進数を使った方が便利です．ですからBASICでも，アドレスの指定にはなるべく16進数を使うようにした方がよいでしょう.

たとえば前に行った〝PEEK　(60000)″などは，〝PEEK　(&HEA60)″と書いた方がよいのです．マシン語で，このアドレスをHLレジスタ(HレジスタとLレジスタのペア)で指定しようとするなら，HレジスタにEAH，Lレジスタに６０Hを入れればよいのですが，こんなとき10進法で60000などと書いてあったのでは2バイトに分けるときたいへんです.

ところで，BASICで&HＦ０００というのをプリントさせてみてください.

```
PRINT &HF000
-4096
Ok
■
```

おや？ 負の数になってしまいましたね．本来ならこれは〝15(＝FH)＊$16^3$″で，61440という数が出るはずです．これには次のようなわけがあります.

CPUはさきほども述べたように，HLレジスタなどを使うと16ビットの数を表すことができます．しかし0と正の数だけではなく，この16ビットの中で負の数も扱いたいという場合があります．そこで次のように考えて，負の数を実現します.

16進数のＦＦＦＦHというのは，16ビットすべてが1という状態です．ここに1を加えるとどうなるでしょうか？

```
    1111 1111 1111 1111
+)                    1
---------------------------
  1 0000 0000 0000 0000
```

このように16のビットはすべて0になり，17ビット目に1が繰り上がります．ところがこの繰り上がった分というのは，レジスタからはみ出してしまうので，結果は0になります．そこでこのFFFFHというのは〝1を加えれば0になる〟ということで，−1という意味に考えることができます．同じようにFFFEHと2，FFFDHと3…というように，2つの数を加えると桁あふれの結果0になるという組がたくさんあります．これらは互いに他方の負の数ということができます．

表2.4を見てください．このように2進数で表したときに，最上位ビット(ビット15)が1のものは負の数，0のものは正の数とされます．

| 2進数 | 16進数 | 10進 | 2進数 | 16進数 | 10進 |
|---|---|---|---|---|---|
| 1 000000000000000 | 8000 | −32768 | 0 000000000000000 | 0000 | 0 |
| 1 000000000000001 | 8001 | −32767 | 0 000000000000001 | 0001 | 1 |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| 1 111111110000000 | FF80 | −128 | 0 000000001111110 | 007E | 126 |
| 1 111111110000001 | FF81 | −127 | 0 00000000 | | 127 |
| ⋮ | ⋮ | ⋮ | 0 000000010000000 | 0080 | 128 |
| 1 111111111111111 | FFFF | −1 | ⋮ | ⋮ | ⋮ |
| └最上位ビットが1(マイナス) | | | 0 111111111111110 | 7FFE | 32766 |
| | | | 0 111111111111111 | 7FFF | 32767 |
| | | | └最上位ビットが0(プラス) | | |

表2.4　2の補数表示

このようにあるビットで表せる数の範囲内で，半分を正の数，残りを負の数として考える方式を**2の補数表示**といいます．BASICで扱う整数というのは2バイトですから，16ビットの2の補数表示でメモリに持っています．

表2.4に示すように，この方式では−32768から＋32767までの整数が表せます．マイナスの方が1つ多いのは，0000Hは0にして，8000Hは負の数にしているからなのです．

ちなみにMSX-BASICでも整数として扱えるのはこの範囲です．前に60000としてそのアドレスにアクセスできたのは，この60000が実数として扱われていたからなのです．

## メモリのRAMとROMの配置

メモリの大きさを表す場合，普通はK(キロ)という単位を使います．これは1024(400H)バイトを1Kバイトとしているので，0H番地からFFFFH(65535)番地までのメモリは64Kバイトの大きさになります．

しかし実際にはそのうちの半分，すなわち0H番地から7FFFH番地までは，MSX-BASICインタープリタのあるROMとして使われています．そのためユーザーが使えるRAMは，残りの32Kバイトになります．これが通常〝ユーザーの使える領域〟とされるので，〝32KのMSX〟といわれるのです．

機械によっては，このRAMが16Kしかないものがあります．これは図2.11のように8000H番地からBFFFH番地にあたる16K分のメモリを付けていない機械です．こういうMSXマシンを使っているときは，この場所にデータやプログラムを入れることはできないので注意してください．

また〝64Kマシン〟というのは，0H番地から7FFFH番地までの分もRAMとして使えるものです．しかし，この場所はインタープリタのはいったROMが切り替わって使われますから，BASICを使っている場合はRAMは32Kマシンと同じ部分しか使えません．

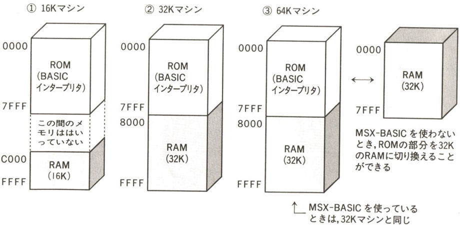



図2.11 MSXマシンのメモリ

# 3章

# マシン語プログラムの作り方

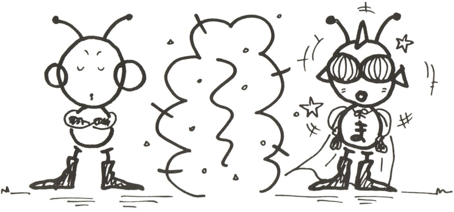
# 3.1

# ニーモニックとコーディング

この章ではマシン語プログラムの作り方を見ていきます．マシン語プログラムの場合，最初にプログラムを考えるときには**ニーモニック**を使ったアセンブリ言語で行います．ニーモニックというのはマシン語の命令１つ１つについて，それぞれに対応する動作を表したものです．マシン語の命令は数字でできているのですが，これを１つ１つ人間が覚えるというのはたいへんなことです．そこでその働きがわかるように表現したものがニーモニックなのです．

作ったプログラムを実際に CPU に実行させるには，これを数字の形の命令にしなければなりませんが，このように数字の形になったマシン語の命令を**マシンコード**といいます．

ニーモニックで書かれたプログラムをマシンコードに変換する作業を，**アセンブル**といいます．この章ではニーモニックの書き方，アセンブルの過程について簡単に説明します．具体的なマシン語の命令は次の４章から見ていきます．



# ニーモニックの形式

ニーモニックは，たいてい次のような形をしています．

| | OPコード部 | オペランド部 |
|---|---|---|
| | | “どこに” “どれを” |
| ① | LD | A ， B |
| | | ディスティネーション ソース |
| ② | LD | （1 2 3 4 H），A |
| | | カッコの中に16ビットの値を書くと，そのアドレスのメモリになる．この場合は1234H番地にAレジスタの内容がはいる．“H”はその数が16進数であることを示す |
| ③ | JP | 4 B 2 0 H |
| | | オペランドにメモリのアドレスを指定する |
| ④ | RET | (OPコードだけでオペランドがない) |

**図3.1 ニーモニックの例**

まず①ですが，これはロード命令といって，レジスタやメモリに値を入れるものです．この場合はAレジスタにBレジスタの値を入れます．このようにニーモニックでは最初に“何をする命令か？”ということを書きます．これを **OP コード**(オペコード)といいます．この例では LD(LOAD の略)がそうです．

そしてその次にはスペースを1つ以上空けて，**オペランド**を書きます．これは，OP コードの命令を“どのレジスタ，メモリのどのアドレスに対して行うか”ということを表すものです．

①の場合“どこに，どれを”というように，オペランドを2つ指定しています．この場合，前のものを**デスティネーション**，後のものを**ソース**といいます．ソースが“どれを”にあたり，デスティネーションが“どこに”にあたります．Z80 の場合，デスティネーション，ソースの順番に書くということを忘れないでください．たとえば“ADD A，B”というのは，“Aレジスタの内容（デスティネーション）にBレジスタの内容（ソース）を加える”という意味になります．

②はデスティネーションがカッコでくくられています．これはメモリであることを表します．この場合Aレジスタの内容をメモリの１２３４H番地に書き込むという意

味になります．このようにオペランドに数値を書く場合は，16進数を使った方がよいでしょう．特にアドレスを指定するときに10進数を使うと，後で不便なことになります．

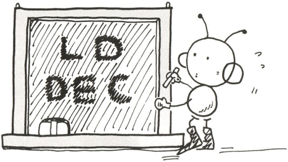
③はジャンプ命令といって，BASICのGOTO文のようなものです．オペランドにはメモリのアドレスを書きます．④はリターン命令です．このように，OPコードだけでオペランドのないものもあります．

ニーモニックを勉強するとき，覚えなければならないOPコード，すなわち命令の数はそんなに多くありません．しかしこのように，きちんと書式が決まっているのでそれに合せた書き方をしなければなりません．

## コーディングの手順

プログラムを作るときは，最初にニーモニックを使ってプログラムを書きます．〝書く〟というのは，とりあえず紙にでも書くのだと思ってください．この作業は**コーディング**といいます．これは自分が作りたいプログラムを，マシン語の命令を使って組み立てていくものです．

コーディングはBASICでプログラムを作る場合にも行うことです．しかしBASICの場合にはコーディングしたものをそのままキーボードから入力すれば，プログラムがメモリにはいり実行できる状態になります．しかしマシン語の場合そうはいきません．なぜならCPUがこのプログラムを実行するには，ニーモニックのままではできないからです．これには，次の節で述べるアセンブルという作業が必要になります．



# 3.2

# アセンブルの方法

前にも述べたように，メモリの中に入れるプログラムというのは数字の形になっています．ですからマシン語も，実際にCPUが実行する時点では数字になっていなければなりません．しかしながらマシン語の場合には，BASICのように文字をそのキャラクタコードに変換するというのではなく，マシンコードに変換しなければなりません．

〝ニーモニックで書かれたプログラムをマシンコードに変換する〞という作業をアセンブルといいます．マシン語のプログラムはアセンブルされて，初めてメモリに入れることができるのです．

ここでアセンブルの例を出してみましょう．**リスト3.1**のようなプログラムがコーディングできたとします．

### リスト3.1

```
LD      A,3
LD      (9000H),A
RET
```

ニーモニックで書かれたソースプログラム．最初は紙の上でこのようなプログラムを書いておく

これをアセンブルしてみます．アセンブルする場合，**リスト3.1**のようにもととなるニーモニックで書いたプログラムを**ソースプログラム**といいます．それに対してアセンブルの結果できたマシンコードのプログラムを，**オブジェクトプログラム**といいます．オブジェクトプログラムは実際にメモリの中にはいってCPUが実行するプログラムのことです．

まず最初に決めなければならないのは，このプログラムがメモリの何番地からはいるかということです．マシン語のプログラムのはいるアドレスは，ちょうどBASICの行番号のようなものですが，BASICのリナンバー(RENUM命令)のように気のきい

た機能はありません．それに後から行の間に命令を入れるなどということもできません．ですからアセンブルする前に，ちゃんとこの先頭アドレスというものを決めておかなければならないのです．

そこでまず最初の命令はＡ０００Ｈ番地からはいるものとします．するとリスト3.2のようになります．

## リスト 3.2

```
 :............マシン語のプログラムのはいる先頭アドレスを決める
A000                        LD       A,3
                            LD       (9000H),A
                            RET
```

マシン語の場合，１つ１つの命令の長さ（何バイトか）が決まっていません．OP コードによっても違いますし，同じ OP コードでもオペランドによって長さが変わったりします．ですから，この場合２番目，３番目の命令が何番地からはいるかということはまだわかりません．最初の命令をアセンブルしてみると，それがメモリを何バイト必要とするため，２番目の命令は何番地から始まる，ということがわかるのです．

１つ１つのニーモニックをアセンブルするときには**ニーモニック変換表***（以下変換表と略）を使います．最初の命令は，変換表のロード命令のところを見て，定数３をＡレジスタに入れる命令は，どのようなマシンコードになるかということを調べればよいのです．この変換表については４章以降でそれぞれの命令を解説するところで説明していきますので，ここでは省略します．

アセンブルが完了すると**リスト** 3.3のようになります．

## リスト 3.3

```
          :.............A000H番地にはいるマシンコード
          :  :....A001H番地にはいる
          :  :    マシンコード
:....A000 3E03                       LD       A,3
:....A002 320090                     LD       (9000H),A
:....A005 C9  ..........A005H番地にはいる   RET
:                       マシンコード
:        オブジェクトプログラム        ソースプログラム
:
各命令ごとにアドレスを書いておく
```

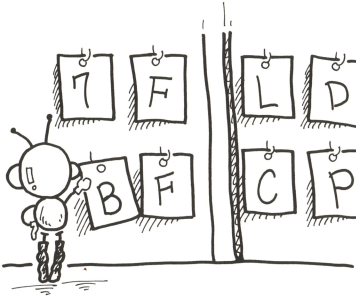
左の２バイトの数字がそれぞれのマシンコードがはいるアドレスで，その右がアセンブルされて実際にメモリに入れられるマシンコードです．この場合Ａ０００Ｈ番地に３ＥＨが，Ａ００１Ｈ番地に０３Ｈがはいることを表します．

ここまでできれば，後は命令を１つ１つメモリに書き込んで，プログラムを実行することができます．

## アセンブラ

このように，変換表を片手にアセンブルする作業を**ハンドアセンブル**といいます．ハンドアセンブルする場合，自分で書いたソースプログラムを自分で１つ１つ翻訳していくわけです．アセンブルという作業は非常にたいへんですから，これをコンピュータに実行させようというのが**アセンブラ**というプログラムです．

アセンブラを使うと BASIC のようにニーモニックをキーボードから打ち込めば，それを自動的にアセンブルさせることができます．




＊ニーモニック変換表は付録を参照

# 3.3

# マシン語のためのツール

さて，ここまででマシン語に関する予備知識が付きましたので，後はどんどん命令を覚えていくことができます．しかし，その前にもう1つだけ問題があります．

たとえばこれから4章以降に出てくるプログラムや，読者の皆さんが自分で作ったプログラムをアセンブルして，それを実行するにはまずMSXのメモリに入れなければなりません．またBASICのRUNにあたる命令を実行しなければマシン語のプログラムは動きません．そこでこの節ではその方法を述べておきます．

MSX-BASICは，残念ながらマシン語を扱うのには不便です．具体的にいうとメモリに直接プログラムやデータを書き込んだり，マシン語のプログラムを実行することが簡単にできません．MSXより高価なコンピュータには**モニタ**といって，メモリにプログラムを書き込んだり，マシン語プログラムを実行するための機能が付いたものがあります．しかしMSXにはこれがないので，何らかの方法でこれを実現しなければなりません．

## 市販ツールについて

MSX用のソフトウェアにはいろいろなものがありますが，その中にはマシン語を扱うためのものもあります．これを買えばもっと楽にマシン語のプログラムを作ったり実行したりすることができます．

こうした市販プログラムにはROMカートリッジ式やカセットテープ式のもの，ディスケット式のものがあります．ディスケットのものは当然ディスクドライブを持っている人でないと使えません．ディスクドライブのない人は，ROMカートリッジやカセットテープのものを使うしかありません．これらのソフトウェアを買うときは，ただメモリを直接アクセスするだけのものもありますが，どうせ買うならアセンブラも

付いたものを買った方がよいでしょう．**表3.1**にMSX用に売られているソフトウェアを示しておきます．

| ソフトウェア名 | メディア | 価格 | 販売元 |
|---|---|---|---|
| モニタ・アセンブラ/グラフィックエディタ | テープ | 4,800 | MIA |
| SIMPLE ASM | ROMカートリッジ | 9,800 | コーラル |
| テープASCII 6月号（1984年） | テープ | 2,800 | ASCII |
| DUAD MSX（発売予定） | ディスク | 30,000(予価) | ASCII |

**表3.1 マシン語ツール**

## BASICモニタのすすめ

次の節ではこういったソフトウェアのない人のために，BASICで書いたモニタプログラムを載せておきます．これはメモリへのアクセスと，マシン語プログラムの実行を行うためのものです．

4章からマシン語のプログラムを作っていきますが，ハンドアセンブルしたプログラム(マシンコード)は，まずメモリに入れなければなりません．またBASICからUSR関数*というものを使ってそのマシン語プログラムを呼び出さなければなりません．

これらをさきほど述べたPOKE文などを使って，1つ1つ処理していたのではたいへんです．ですからこの2つの操作(プログラムをメモリに入れる＋実行する)を簡単にできるようにしたものがこのプログラムです．

4章以降のプログラムは，市販のソフトウェアがないと動かせないというのでは不公平になります．ですからこの本に出てくるプログラムは，すべてこのモニタプログラムがあれば動かせるようにできています．MSXを持っている人は，必ずこのプログラムを入力して，セーブしておいてから4章以降を読むようにしてください．




＊ USR関数については15章参照

# 3.4

# BASICモニタの使い方

まずリスト3.4の〝BASICモニタ〟(以下モニタと略)を入力してください．このプログラムは，4章以降でマシン語のプログラムをMSXで実行するのに使うプログラムです．これには3つのコマンドがありますから，その使い方をこの節で説明します．

## リスト 3.4

```
1000 ' MONITOR PROGRAM
1010 CLEAR 300,&HCFFF
1020 SCREEN 0 : CLS : KEY OFF: COLOR 1,15
1030 'FUNCTIONS
1040 DEF FNA$(X)=RIGHT$("000"+HEX$(X),4)
1050 DEF FND$(X)=RIGHT$("0"+HEX$(PEEK(X)),2)
1060 DEF FNNH(A$)=INSTR("0123456789ABCDEFabcdef",A$)=0
1070 PRINT " BASIC MONITOR FOR MSX"
1080 ' START
1090 PRINT
1100 LINE INPUT ">";BU$
1110 IF BU$="" THEN 1100
1120 CO$=LEFT$(BU$,1)
1130 IF INSTR("Dd",CO$) THEN 1170
1140 IF INSTR("Ss",CO$) THEN 1410
1150 IF INSTR("Gg",CO$) THEN 1530
1160 PRINT "ILLEGAL COMMAND" ; GOTO 1100
1170 ' DUMP MEMORY
1180 '  (D コマンド)
1190 IF LEN(BU$)=1 THEN ME=MS+63 : GOTO1290
1200 J=INSTR(BU$," ") : IF J=0 THEN 1280
1210 IF J-4>2 THEN K=J-4 ELSE K=2
1220 L=LEN(BU$)-J : IF L>4 THEN L=4
1230 MS=VAL("&H"+MID$(BU$,K,J-K))
1240 IF MS<0 THEN MS=MS+65536!
1250 ME=VAL("&H"+RIGHT$(BU$,L))
1260 IF ME<0 THEN ME=ME+65536!
```

1010行: 32K，64Kシステムのマシンのときは，&H8FFFとすると，9000H番地以降が使える

1060行: 16進数の文字であるか

1130〜1150行: コマンドは小文字でも大文字でもよい

次頁へ続く

```
1270 GOTO 1290
1280 GOSUB 1600 : MS=MM : ME=MS+63
1290 ' DUMP FROM MS TO ME
1300 FOR I=MS TO ME STEP 8
1310  PRINT FNA$(I);" ";
1320  FOR Q=0 TO 7
1330   PRINT FND$(I+Q);" ";
1340   GOSUB 1650
1350   IF I+Q=ME THEN Q=8
1360  NEXT
1370  PRINT
1380 NEXT
1390 MS=ME+1
1400 GOTO 1080
1410 ' SET MEMORY
1420 '  (S ｺﾏﾝﾄﾞ)
1430 IF LEN(BU$)=1 THEN 1450
1440 GOSUB 1600 : SS=MM
1450 PRINT FNA$(SS);" ";
1460 PRINT FND$(SS);"-";
1470 GOSUB 1720
1480 IF ID$=" " THEN 1510
1490 IF ID$="." THEN PRINT "."; : GOTO 1080
1500 POKE SS,VAL("&H"+RIGHT$(ID$,2))
1510 SS=SS+1 : PRINT " ";
1520 IF SS=INT(SS/4)*4 THEN PRINT : GOTO 1450 ELSE 1460
1530 ' EXECUTE
1540 '  (G ｺﾏﾝﾄﾞ)
1550 IF LEN(BU$)=1 THEN PRINT "WHAT?" : GOTO 1080
1560 GOSUB 1600
1570 DEF USR=MM       } マシン語プログラムの実行
1580 DUMMY=USR(0)     }
1590 GOTO 1080
1600 ' EVALUATE ARGUMENT
1610 M=LEN(BU$)-1 : IF M>4 THEN M=4
1620 MM=VAL("&H"+RIGHT$(BU$,M))
1630 IF MM<0 THEN MM=MM+65536!
1640 RETURN
1650 ' PRINT CHARACTER
1660 CX=POS(0)
1670 LOCATE 30+Q,CSRLIN
1680 AD=PEEK(I+Q) : IF AD=&H7F THEN AD=&H2E
1690 IF AD<&H20 THEN PRINT "."; ELSE PRINT CHR$(AD);
1700 LOCATE CX,CSRLIN
1710 RETURN
1720 ' GET 1 BYTE ……………Sコマンドの中での1バイト入力
1730 A$=INPUT$(1) ……………1文字目入力
1740 IF A$=" " THEN PRINT "  "; : ID$=" " : RETURN
```

次頁へ続く

```
1750 IF A$="." THEN ID$="." : RETURN
1760 IF FNNH(A$) THEN 1730
1770 PRINT A$;
1780 B$=INPUT$(1) ……………2文字目入力
1790 IF B$=CHR$(8) THEN PRINT B$;" ";B$; : GOTO 1730 ……………
1800 IF FNNH(B$) THEN 1780
1810 PRINT B$;                         バックスペースなら,1文字目の入力やり直し
1820 ID$=A$+B$
1830 RETURN
```

## コマンドの種類

このモニタにあるコマンドは次の3つです.

① D(ダンプメモリ):メモリの内容を表示する
② S(セットメモリ):メモリの内容を書き換える
③ G(ゴー)　　　　:マシン語プログラムにジャンプ(実行)する

モニタではマシン語という〝危険な〟ものを扱います.ときにはマシン語のプログラムが暴走して手が付けられなくなるということも考えられますので,プログラムを入力したら必ずセーブしておいてください.このプログラムの中では,メモリのアドレスやデータを表示したり入力したりするときは,16進表示で行います.プログラムをRUNさせると,

BASIC MONITOR FOR MSX

というメッセージとともに〝>〟というプロンプトが出て,その右にカーソルが表示されます.これがこのモニタの〝コマンド待ちの状態〟で,上にあげた3種類のコマンドを実行することができます.以下コマンドの使い方を1つずつ説明しましょう.



# 各コマンドの使い方

**▶Dコマンド**

Dコマンドは〝今メモリの中にはどんなデータがはいっているか？〟を表示するコマンドです．ですからこれを使えばメモリの内容を見ることができるのです．このコマンドの書式は次のとおりです．

① Dｘｘｘｘ　ｙｙｙｙ

② Dｘｘｘｘ

③ D

```
>DD800 D81F
D800 FF FF FF FF FF FF FF FF  ■■■■■■■■
D808 FF FF FF FF FF FF FF FF  ■■■■■■■■
D810 FF FF FF FF FF FF FF FF  ■■■■■■■■
D818 FF FF FF FF FF FF FF FF  ■■■■■■■■

>DDC00
DC00 FF FF FF FF FF FF FF FF  ■■■■■■■■
DC08 FF FF FF FF FF FF FF FF  ■■■■■■■■
DC10 FF FF FF FF FF FF FF FF  ■■■■■■■■
DC18 FF FF FF FF FF FF FF FF  ■■■■■■■■
DC20 FF FF FF FF FF FF FF FF  ■■■■■■■■
DC28 FF FF FF FF FF FF FF FF  ■■■■■■■■
DC30 FF FF FF FF FF FF FF FF  ■■■■■■■■
DC38 FF FF FF FF FF FF FF FF  ■■■■■■■■

>D
DC40 00 00 00 00 00 00 00 00  ........
DC48 00 00 00 00 00 00 00 00  ........
DC50 00 00 00 00 00 00 00 00  ........
DC58 00 00 00 00 00 00 00 00  ........
DC60 00 00 00 00 00 00 00 00  ........
DC68 00 00 00 00 00 00 00 00  ........
DC70 00 00 00 00 00 00 00 00  ........
DC78 00 00 00 00 00 00 00 00  ........

>■
```

D800H番地からD81FH番地までを表示

yyyyを指定しないと64バイトが表示される

64バイト

DC3FH番地の次のアドレス

xxxxも指定しないと前に行ったDコマンドの最後の番地の次から表示する

64バイト

左側の16進数に対応するキャラクタコードが表示される

**図3.2 Dコマンドの実行例**

〝xxxx″とか〝yyyy″というのは4桁の16進数で表したアドレスです．①のように入力すると，最初に書いたアドレスから2番目に書いたアドレスまでのメモリ内容を表示します．このときxxxxとyyyyの間にはスペースを入れますが，Dの後ろxxxxの前にはスペースを入れず，詰めて入力します．

②のように2番目のアドレスを入れないと，自動的にxxxx＋63(3FH)になり，64バイトが表示されます．また③のように最初のアドレスを指定しないと，前に実行したDコマンドの終了アドレス(yyyyのこと)の次のアドレスになります．ですからDだけを何度も実行すると，連続したメモリの内容を64バイトずつ区切って見ることができます．

Dコマンドを実行すると，左に先頭アドレス(xxxxのこと)が出て，その右に8バイトずつメモリの内容が表示されます．その右の8つの文字は，それぞれメモリの中の1バイトが何という文字のコードであるかを表しています(図3.2)．

### ▶Sコマンド

これはメモリにデータを書き込むコマンドで，マシン語で作ったプログラムを入力するのに使います．このコマンドの書式は次のとおりです．

① Sxxxx
② S

Dコマンドと同じように4桁の16進数でアドレスを指定します．ただしSコマンドの場合はアドレス指定は1つだけです．②のようにアドレスを指定しないと，前のSコマンドでセットし終わったメモリの次のアドレスが選ばれます．

このコマンドを実行すると，指定したアドレスといまそこにはいっているデータが画面に表示されて，その右にカーソルが移ります．ここで2桁の16進数を入れるとそのメモリに書き込まれ，自動的に次のメモリに移っていまのデータとカーソルが表示されます．入力できるデータは16進数だけですから，0～9までの数字とAからFまでの英文字で入力します(それ以外の文字は受け付けません)．

もし，いまあるデータを書き換える必要がなければ，スペースキーを押します．すると書き換えを行わずに次のアドレスに進みます．Sコマンドを終了したいときは〝.″(ピリオド)を押します．するとコマンド待ちの状態に戻ります*(図3.3)．

```
>SD000
D000 FF-90 FF-21 FF-   FF-E3
D004 FF-B1 FF-   FF-3A FF-80
D008 FF-.
>■
```

アドレス
前にそのアドレスにはいっていた内容
新しく入れたい値を入力する
書き換えないときはスペースキーを1回押す
スペースキーを押す
ピリオドを押すとSコマンドから抜ける(コマンド待ちの状態に戻る)

図3.3 Sコマンドの実行例

▶Gコマンド

これは，メモリにSコマンドで入れたマシン語のプログラムを実行するためのコマンドです．このコマンドは

① G x x x x

というように，ジャンプする先のアドレスを指定します．このコマンドは正しく入力したマシン語プログラムの，正しい先頭アドレスにジャンプしないと，間違いなく暴走すると思ってください．

Gコマンドが正しく動いているかどうかを確かめるには次のようにします．

まずSコマンドでE000H番地にC9Hを書き込みます．1バイトだけで結構です．そうしたらSコマンドから抜け，次に〝GE000〟を実行します．これでまた〝>〟が出てコマンド待ちの状態に戻れば成功です(図3.4)．

```
>SE000 ⏎
E000 E5-C9 28-.
>GE000 ⏎ ……………Gコマンドの実行

>■ ……………またカーソルが出たらモニタプログラムは正常に動いている
```

C9Hを書き込む

図3.4 Dコマンドの実行例

＊もしDコマンド，Sコマンドを実行した結果うまく動いたら，次のGコマンドをする前にテープにセーブしておいてください．

## マシン語開発ツール

3章では，マシン語開発ツールとしてBASICで作成したモニタを紹介したが，やはり本格的にマシン語のプログラムを作るとなると，専用の開発ツールが欲しいところだ．

マシン語の開発ツールとしては，アセンブラのほかにディスアセンブラ，デバッガ，トレーサといったものがあり，こういったものを使いこなすことができれば，マシン語プログラムの開発効率は格段に上がるようになる．

ディスアセンブラというのは，アセンブラの反対の作業を行うツールで，メモリに記憶されているマシン語プログラムをニーモニックに変換する．これがあると，自分で作ったプログラムばかりでなく，市販されているプログラムや雑誌に掲載されているダンプリストの解析に役立てることができる．デバッガはBASICモニタの機能をさらに拡張したものと考えればよい．マシンコードのメモリへのセット，メモリのダンプ，マシン語プログラムの実行のほかに，I/Oポートをアクセスしたり，レジスタの内容を表示するといったことが簡単にできる．トレーサは，マシン語のプログラムを1命令ごとに実行し，同時にその結果も出力するもので，プログラムの誤りを容易に見つけることができるものだ．

こういったプログラムは，市販されているものを利用するのがいちばん手軽だが，自分で作ってみるのも結構楽しいものだ．マシン語に習熟してきたら，一度はチャレンジしてみるのもよいのではないだろうか．



# 4章

# メモリとレジスタのデータ転送

# 4.1

# ロード命令の働き

さて，これからいよいよマシン語の命令を覚えていきます．マシン語を覚えていく際，まず最初に必要になるのが，ここで説明する**ロード命令**です．これは，レジスタやメモリにデータを書き込む操作をするものです．

どうしてこれが最初に必要になるかというと，計算をするにしてもまず最初はレジスタに計算をする値を持ってくるということが必要になるからです．つまりあらゆるマシン語の命令の基本になるのがこのロード命令なのです．

この章はマシン語の勉強の最初にあたる部分です．ですからロード命令そのものも大切ですが，次にあげることも大切ですから，気を付けて読んでいってください．

- ○ **各レジスタの役割**
- ○ **モニタの使い方**

## ロード命令の働き

ロード命令は，BASICでいうと代入文のような働きをします．たとえば〝X=0″とすると，Xという変数に0という値がはいりますが，これがマシン語のロード命令の場合には変数ではなく，レジスタやメモリにデータを入れるという命令になります．

たとえば何か足し算をしようと思ったら，まずレジスタに足される数を入れ，それからそのレジスタに足す数を加えるということを行います(演算命令は9章参照)．BASICならば代入文を1つ書けばすむところも，マシン語の場合にはこのように計算の過程も細かくプログラムを作ってやらなければなりません．

このように，あるところから別のところにデータを転送することを総称して〝ロ

ード(LOAD)〟といい，〝Aレジスタに3という値をロードする〟とか〝メモリにBレジスタの内容をロードする〟というように使います．




1つの命令でロードできるデータは，8ビットまたは16ビットです．これはZ80のレジスタが8ビットか16ビットであるためです．メモリ1番地には8ビットのデータをロードすることができます．

**図4.1**はロード命令が〝どこに何をロードできるか〟を表しています．レジスタは通常相互にデータをロードすることができます．またプログラムの最初にはレジスタに指定した値を入れるのがふつうですが，図の〝定数〟というのがその意味です．〝Aレジスタに45をロードする〟というように使います．

またレジスタとメモリは，相互にデータをやり取りすることができます．これはCPUがメモリをアクセスするということを意味します．このようにメモリをアクセスするときには必ずアドレスを指定します．

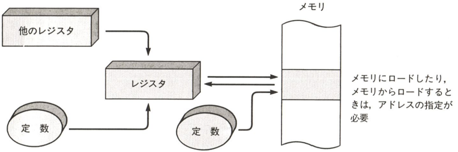



図4.1 8ビットロード命令の種類

Z80のレジスタには8ビットのものと16ビットのものがあります．ですからロード命令も，一度に8ビットを扱うものと16ビットを扱うものがあります．次の節からはこの8ビットロード命令と16ビットロード命令を分けて説明していきます．

# 4.2

# 8ビットロード命令

8ビットロード命令は，8ビットのデータをあちこちに転送(ロード)する命令です．8ビットのデータというのは，8ビットレジスタ(アキュムレータと汎用レジスタ)の内容やメモリに記憶されている8ビットの値を意味します．

ここでは8ビットのデータ(0から255までの数値)以上の数は扱えませんから注意してください．

## ニーモニックの書き方

8ビットロード命令をニーモニックで書くと次のようになります．

①ＬＤ　　Ａ，９……Aレジスタに9をロードする
②ＬＤ　　Ｂ，Ｈ……BレジスタにHレジスタの値をロードする

このようにニーモニックを書く場合は，OPコードに〝LD(ロード)〟を書いてオペランドを2つ指定します．最初のオペランドがデスティネーションといってデータの転送先，次のオペランドがソースといってデータの転送元を書きます．上の2つのロード命令のデスティネーションは，どちらも8ビットレジスタになっていますが，このようにレジスタを指定する場合は〝A〟，〝B〟というようにレジスタの名前を書きます．

①はAレジスタに9という定数を入れる命令です．②はHレジスタの内容をBレジスタにロードするという意味になります．この場合，最初にBレジスタにあったデータはなくなって，Hレジスタの内容がBレジスタにはいります．Hレジスタの内容はもちろんそのままです．

これらが8ビットのレジスタにデータを入れるロード命令です．ここで何かプログラムを作ってみたいところですが，これだけでは作れません．なぜならモニタからは

直接レジスタの内容を見ることができませんから，レジスタにデータを入れただけではそれを確かめることができないからです．

そこで，モニタを使ってプログラムを作るときには，レジスタの内容をメモリに入れる必要があります．データをメモリに入れさえすれば，モニタでそれを確かめることができるのです．次の③から⑥はメモリに関するロード命令です．

```
③LD    A,(9898H)
④LD    (3943H),A
```

③はメモリの９８９８Ｈ番地からＡレジスタにデータをロードする命令です．このように16ビットの値をカッコでくくってやると，それはカッコの中の値をアドレスとするメモリという意味になります．④は逆にＡレジスタの内容をメモリの３９４３Ｈ番地にロードします．

これはちょうどBASICのPEEK関数とPOKE文にあたります．このようにメモリのアドレスを直接オペランドに指定してメモリにアクセスするときは，必ずＡレジスタでなければなりません．なお③や④の〝A〞のところにＢレジスタやＣレジスタを書いたり，④のソースに〝6〞というような定数を書くことはできません．

このようにＡレジスタ以外のレジスタや定数を使いたいときは，⑤のような方法をとらなければなりません．これは16ビットのHLレジスタを使ってメモリのアドレスを指定しています．たとえばこれを実行したときのHLレジスタの内容が８９０９Ｈならば，８９０９Ｈ番地の内容をＤレジスタにロードします．このようにHLレジスタでアドレスを指定するといろいろなロードが可能になります．⑥のように直接定数をメモリにロードすることもできます．もちろんこの命令を実行する前には，HLレジスタにアドレスの値をロードしておく必要があります．

```
⑤LD    D,(HL)
⑥LD    (HL),4
```

ちなみに16ビットレジスタをメモリのアドレスとして使うロード命令として，⑦のようなものがあります．このようにBC，DEレジスタを使った場合は，これまたＡレジスタでしかやり取りができなくなります．

```
⑦LD    (BC),A
```

## ニーモニックを書くときの注意

ところで，ここにあげたように，ニーモニックの中にはレジスタ名のほかに定数の数字をそのまま書くこともできます．この場合の注意ですが，定数を書くときには 10 進数でも 16 進数でもかまいません．しかし書いた値がそのどちらであるかは区別できないと困りますから，たいてい次のようなルールがあります．

- ○ 10 進数はそのまま書く
- ○ 16 進数を書く場合は最後に〝H〞を付ける（例：２９Ｈ，１ＤＨ）
- ○ 16 進数で最初の桁がＡ～Ｆの場合にはその前に０を付ける（例：０Ｃ３Ｈ）

これは市販されているアセンブラの文法ですから，アセンブラを使う場合には必ず守らなければなりません．自分でハンドアセンブルする場合には必ずしも守る必要はありませんが，プログラムが大きくなるとわかりにくくなるので，なるべくこのように統一しておく方がよいでしょう．

## マシンコードの求め方

ニーモニックを書いたら，それをハンドアセンブルによってマシンコードに変換します．ここではハンドアセンブルも初めてなので，その方法を詳しく説明していきます．

ニーモニックにはそれぞれ１つずつ 16 進数のマシンコードが割り当てられます．たとえば，

```
LD    A,A  は  7FH
LD    A,B  は  78H
```

という具合です．つまり〝LD　A，B〞というニーモニックは，実際にはメモリに７８Ｈという数を入れなければなりません．CPU がこの７８Ｈというマシンコードを実行するとＢレジスタの内容をＡレジスタにロードするのです．

８ビットのレジスタ間のロード命令は，すべて１バイト命令になります．しかし，

これをすべて覚えるのはたいへんです．実際には変換表(ニーモニックとマシンコードの対応が書いてある表：付録参照)がありますから，その見方さえわかっていればよいのです．ここでこの変換表の見方を説明します．

**表**4.1の左側にOPコードである〝LD″と，デスティネーション，上にはソースが書いてあります．この2つが交わるところにそのマシンコードが書いてあるのです．たとえば，これで

LD　　B，L

というニーモニックを変換したい場合，〝LD　B，×″と〝L″の交差するところを見て，45Hという数を得ます(変換表の中の数はすべて16進数)．

| × | A | B | C | D | E | H | L | (HL) | (BC) | (DE) | (IX+d) | (IY+d) | (nn) | n |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| LD A，× | 7F | 78 | 79 | 7A | 7B | 7C | 7D | 7E | 0A | 1A | DD<br>7E<br>d | FD<br>7E<br>d | 3A<br>n<br>n | 3E<br>n |
| LD B，× | 47 | 40 | 41 | 42 | 43 | 44 | 45 | 46 | | | DD<br>46<br>d | FD<br>46<br>d | | 06<br>n |
| LD C，× | 4F | 48 | 49 | 4A | 4B | 4C | 4D | 4E | | | DD<br>4E<br>d | FD<br>4E<br>d | | 0E<br>n |
| …d)，× | …<br>77<br>d | …<br>70<br>d | …<br>71<br>d | …<br>71<br>d | …<br>d | d | | | | | | | | …n |
| LD (nn)，× | 32<br>n<br>n | | | | | | | | | | | | | |

本文中で参照する変換表は必要な部分のみを取り出していますので，他の命令を調べる場合は付録を参照してください．

**表4.1 8ビットロード命令変換表**

ロード命令の中には，オペランドに8ビットの定数や16ビットの定数でアドレスを指定したものがくる場合があります．この場合はマシンコードがオペランドの分だけ長くなります．たとえば，

LD　　A，9………3E　09

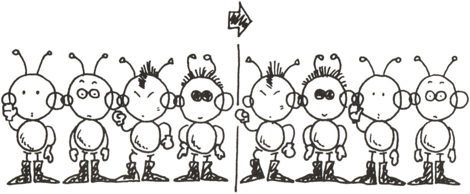
という命令は，〝3E，09″という2バイトの命令になります．1バイト目の3EHはOPコードにあたる部分で，〝次の数をAレジスタに入れよ″という意味の命令になり，09Hがその定数になります．変換表を見ると，〝3E　n″というように書いてあります．これはnに8ビットの数なら何を書いてもよいということです．

次にメモリのアドレスを指定する場合を見てみましょう．この場合は，定数が2バイトになります．たとえば，

```
LD    (8248H), A………32  48  82
       ↑ ↑
      上位 下位
```

という命令は，〝32，48，82〞という3バイト命令で，最初の32HがOPコードで，48Hと82Hがオペランドです．

ここで，16ビットの値の上位の82Hと，下位の48Hがひっくり返っていることに注意してください．つまり指定するアドレスは8248Hなのですが．これを1バイトずつメモリに入れるときは，下位，上位の順に〝48，82〞と書かなければなりません．変換表には〝nn〞というように上位，下位が区別されていないので，ニーモニックを変換するときに注意してください．



## Challenge

**▶メモリにデータを入れる**

ニーモニックの変換の方法もわかったところでさっそく実際にプログラムを作ってみましょう．BASICモニタの用意はOKですか？ もう一度モニタのコマンドを復習しておきましょう．S(セット)，D(ダンプ)それにG(ゴー)の3種類です．

それではこれまでに出てきた8ビットロード命令を使って，プログラムを作って実行してみましょう．最初は非常に短いプログラムから始めます．

## テーマ　D800H番地に8という数を入れる

ロード命令には〝指定したアドレスに指定した値をロードする〃という命令はありませんから，いったんAレジスタに8をロードして，それをD800H番地に入れるか，またはHLレジスタにアドレスのD800Hを入れておいて，そのメモリに8をロードするというプログラムになります．

最初の方法でプログラムを作ってみると**リスト4.1**のようになります．

### リスト 4.1

```
LD     A,8 ..............................Aレジスタに定数8をロードする
LD     (0D800H),A .............Aレジスタの内容(8) をD800H番地にロードする
```

次にこれをアセンブルするわけですが，その前にプログラムがはいるアドレスを決めなければなりません．この場合D800H番地には8というデータを入れますから，プログラムはD000H番地から入れることにしましょう．変換表を見ながらこれをハンドアセンブルすると**リスト4.1A**のようになります．

### リスト 4.1A

```
アドレス マシンコード                          ニーモニック
D000  3E08                            LD    A,8
D002  3200D8 ......D800Hの上位，下     LD    (0D800H),A
                   位を逆にしてメモ
                   リに入れる
D005  C9                              RET ..............モニタのコマンド待ちの状態に戻る
```

まず最初のニーモニックを変換すると2バイト必要になることがわかります．これはD000H番地とD001H番地にはいり，次のメモリへロードする命令はD002H番地からはいります．これは3バイト命令で，そこからD004H番地までのメモリが必要になります．このようにハンドアセンブルする場合には命令を1つ1つアセンブルしていって，その結果次の命令のはいるアドレスがわかるようになるのです．

最後にRETという命令があります．これはBASICのRETURN文に当たる命令です．詳しい説明は6章で行いますが，モニタのGコマンドでこのプログラムを実行

したとき，プログラムの最後にこの命令を加えておけば，実行が終わったとき，モニタのコマンド待ちの状態に戻ってきます．

このRETがないとマシン語のプログラムはメモリへのロードの後，次のメモリ(**リスト4.1A** ではD005H番地)の命令を読み出しに行って，いつまでもモニタに戻ってきません．つまりそのまま暴走してしまいます．ですからマシン語のプログラムの終わりには必ずRETを入れて，モニタのコマンド待ちの状態に戻るようにするのです．

ハンドアセンブルが終わったら，できたマシンコードをモニタのSコマンドで入力していきます．D000H番地から順に3E，08，32，00，D8そしてD005番地にC9を入れます．**図4.2** がSコマンドでこのプログラムを入力したところです．

```
>SD000 ⏎ ……………SコマンドによってメモリのD000H番地からマシンコードを1バイトずつ入力していく
D000  00-3E  00-08  00-32  00-00
D004  00-D8  00-C9  00-.
>■                     └─Sコマンドを終了する
```

**図4.2 マシンコードの入力**

6バイトだけですから間違いなく入力できると思いますが，いちおうDコマンドで確かめてみましょう．

```
>DD000  D00F ⏎ ……………D000H番地からD00FH番地のメモリの内容を表示する
D000┌3E 08 32 00 D8 C9 00 00   >.2.リノ..
D008│00 00 00 00 00 00 00 00   ........
    │             └─D006H番地以降には無意味な数字が書き込まれている
>■  └─D000H番地からD005H番地にはモニタのSコマンドで入力したマシンコードが書き込まれている
```

**図4.3 プログラムの確認のようす**

このようにマシン語のプログラムを入力するときは，Sコマンドでメモリに入れてDコマンドで確かめるようにしてください．6バイトだけだからといって甘く見てはいけません．マシン語は1バイト間違っても暴走してしまう恐れがありますから，実行(Gコマンド)するのはまったく間違いがないとわかった最後の最後にします．

うまく入力できたらGコマンドで実行してみましょう．このプログラムの先頭(プロ

グラムの始まる番地)はＤ０００Ｈですから，〝ＧＤ０００″とタイプしてリターンキーを押します．

Ｇコマンドを実行すると，すぐに画面にはプロンプト〝＞″とカーソルが表示されます．これはマシン語のプログラムが終わったということです．これは本当にアッという間に終わるはずです．もし実行してもモニタに戻らずにMSXが黙ってしまったら，これはもう暴走しています．お気の毒ですが，電源を切ってもう一度モニタプログラムをロードしてください．

さてうまくモニタに戻ってきたら，Ｄコマンドで目的が達成できたかどうかを確かめてみましょう．**図4.4**のようにうまくいきましたか？ Ｄ８００Ｈ番地に〝０８″がはいっていればメデタシメデタシです．うまくいかなかった人は，もう一度打ち込んだプログラムをよく見直してください．

```
>GD000 ⏎ ……………Gコマンドによって D000H番地から入力したマシン語プログラムを実行する

>DD800 D807 ⏎ ……………結果を確める
D800 08 FF FF FF FF FF FF FF    ■■■■■■■■
      ↑
      └─いまのプログラムで入れた値
>■
```

**図4.4 リスト4.1実行例**

このプログラムはオペランドの部分だけを書き換えることができます．Ｄ００１Ｈ番地のデータを書き換えればメモリに入れる値が変わりますし，Ｄ００３Ｈからの２バイトを変えれば値を入れるメモリのアドレスが変わりますので，いろいろと試してみてください．

### ▶ HLレジスタを使う方法

今度は同じことを別の方法で試してみます．さきほど述べたHLレジスタを使う方法でプログラムを書くと**リスト4.2**のようになります．

## リスト4.2

```
LD      HL,0D800H  ……………HLレジスタにアドレス(D800H)をロードする
LD      (HL),8     ……………HLレジスタの内容が示すアドレスに定数8をロードする．
                           この場合，D800H番地のメモリに8がはいる
```

これもさきほどと同様にＤ０００Ｈ番地からプログラムを入れてみましょう.

## リスト 4.2A

```
D000 2100D8          LD      HL,0D800H
D003 3608            LD      (HL),8
D005 C9              RET ……………リスト4.1Aと同様にプログラムの最後には
                                RETを入れておく
```

これをさきほどと同じように入力して，実行して結果を見ているのが**図4.5**です.

```
>SD000 ⏎
D000 FF-21 FF-00 FF-D8 FF-36 ⎫
D004 FF-08 FF-C9 FF-.        ⎭プログラムの入力
>DD000 D007 ⏎
D000 21 00 D8 36 08 C9 FF FF   !.リ6.ノ■■ ……………プログラムの確認

>GD000 ⏎ ……………実行

>DD800 D807 ⏎ ……………実行結果を確める
D800 08 FF FF FF FF FF FF FF   .■■■■■■■

>■
```

**図4.5 リスト4.2の入力から実行結果の確認**

このように，同じことをするにもいろいろな方法がありますが，なかでもここで実行したような，メモリに値をロードするというのは重要な作業です．たとえば後でいろいろ計算するプログラムが出てきますが，計算はCPUの内部で行うので，結果はレジスタに残ります．ところがレジスタの内容はBASICではのぞくことができません．そのためにはメモリに入れておいて，BASICでPEEK関数によって読み出してやらなければなりません．

マシン語で，8ビットの値をメモリにロードするには次のような方法があります．

**①Aレジスタに値を入れて，アドレスを指定してロードする**

**②16ビットレジスタでアドレスを指定して，そこにロードする**

どちらもこれから何回も使いますから，よく覚えておいてください．

# 4.3

# 16ビットロード命令

16 ビットロード命令は，一度に 16 ビットのデータを扱います．たとえば 16 ビットレジスタにデータをロードしたり，メモリに 2 バイトのデータを読み出したり書き込んだりします．

16 ビットロード命令で扱うレジスタは，B, C, D, E, H, L という 8 ビットの汎用レジスタを 2 つずつつないで BC, DE, HL というようにしたもの，それに 16 ビット専用の IX, IY, SP です．

## 16ビットロード命令の制限

16 ビットロード命令は，8 ビットロード命令に比べると種類が少なくなります．どういうロードができるかを全部あげてみると**図 4.6** のようになります．

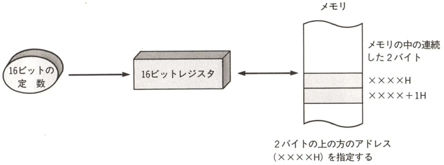



**図4.6 16ビットロード命令の種類**

8ビットのロード命令と比べてみると，まず16ビットレジスタの値を他の16ビットレジスタに入れるという命令がありません*．ですから，たとえばBCレジスタの内容をHLレジスタに入れたいときは，

```
LD    H,B
LD    L,C
```

という2つの8ビットロード命令を使います．16ビット専用のIXやIYというレジスタに他のレジスタから値を入れたいときは，7章で述べるPUSH／POPという命令を使います．

またメモリに，16ビットのデータを一度に書き込んだり読み出したりするときには，必ずアドレスを定数で指定します．これも(HL)というように，レジスタでアドレスが指定できればよいのですが，残念ながらこういう命令はありません．

## ニーモニックの書き方

16ビットのロード命令のニーモニックはどう書くかを見てみましょう．次の①から③のニーモニックは，どれも16ビットレジスタに定数をロードするものです．まず8ビットの場合と同じように，デスティネーションにレジスタ名(もちろん16ビットレジスタ)，ソースに16ビットの定数を書きます．

```
①LD    DE,8945H
②LD    BC,9
③LD    IX,9435H
```

①を実行するとDレジスタに上位8ビットの89Hが，Eレジスタに下位8ビットの45Hがはいります．

次の②の場合も，ソースの9という定数は16ビットと考えます．この場合上位8ビットは00Hと考え，Bレジスタには00Hが，Cレジスタには09Hがはいります．③はIXレジスタに9435Hという定数がロードされます．図4.7はこれらの命令の動作を示したものです．

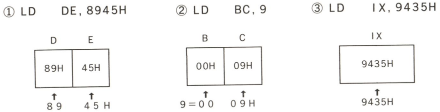



図4.7 ①〜③の動作

次にメモリと16ビットレジスタの間でデータをやり取りするものを見てみます.

④ LD (0A000H), BC
⑤ LD HL, (9000H)

ニーモニックの書き方は，8ビットの場合とまったく同じです．ただ注意しなければならないのは，これらの命令がどう実行されるかということです(図4.8).

④の場合は, A000H番地からの2バイトにBCレジスタの内容を入れます．その際，上位のBレジスタの内容はA000H番地ではなくA001H番地にはいり，下位のCレジスタの内容がA000H番地にはいります．⑤の場合も同じで，Hレジスタには9001H番地の内容，Lレジスタには9000H番地の内容がはいります.

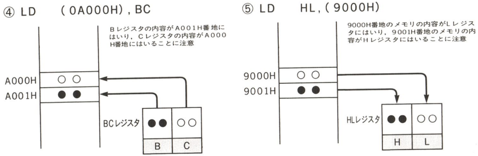



図4.8 ④〜⑤の動作

---

＊ SPというレジスタにHL, IX, IYレジスタの内容を入れるという命令はあるのですが，他のレジスタには用意されていません．ただしSPはあまり値を入れたりするものではありません(暴走のもとですからやらないでください).

## マシンコードの求め方

次に16ビットロード命令のマシンコードを見ていきましょう．16ビットロード命令は，ふつう転送するデータか，メモリのアドレスのどちらかを2バイトの定数で指定します．これに最初の命令のコード1バイトを加えると，一般に16ビットロード命令は3バイト命令になります．

まず定数をレジスタに入れる場合を見てみましょう．

```
LD    HL,8945H…………21  45  89
LD    BC,8000H…………01  00  80
```

最初の21Hとか01Hというのが，命令のOPコードの部分でその後の2バイトが，実際にレジスタに入れる値になります．この場合もこれまでと同じように，16ビットのデータは上位と下位を逆にします．

またメモリのアドレスを指定する場合も同じで次のようになります．

```
LD    HL,(8098H)………2A  98  80
LD    (0A994H),BC……ED  43  94  A9
```

上のように，BCレジスタ(DEレジスタでも同じ)を使う場合は，OPコードが2バイトになりますから注意してください．16ビットロード命令の場合にはそんなに数も多くありませんが，16ビットのデータをオペランドにする場合は上位と下位を逆にするということを覚えておいてください．

16ビットロード命令の変換表は**表4.2**のようになっています．

これも見方は8ビットの場合と同じです．たとえば〝LD　BC,(8000 H)〟の命令を見たい場合は，〝LD　BC,×〟と(nn)が交差するところを見ます．するとこれはOPコードの部分が2バイトで，〝ED，4B〟であることがわかります．

その下のnnというのは，ここに16ビットの値を書くという場所です．この表にはnnとなっていますが，実際に入れるときには下位，上位の順にしなければなりません．結局この命令は〝ED，4B，00，80〟という4バイト命令になります．

| × | BC | DE | HL | SP | IX | IY | nn | (nn) |
|---|---|---|---|---|---|---|---|---|
| LD BC, × | | | | | | | 0 1<br>n<br>n | E D<br>4 B<br>n<br>n |
| LD DE, × | | | | | | | 1 1<br>n<br>n | E D<br>5 B<br>n<br>n |
| LD HL, × | | | | | | | 2 1<br>n<br>n | 2 A<br>n<br>n |
| LD SP, × | | | F 9 | | D D<br>F 9 | F D<br>F 9 | 3 1<br>n<br>n | E D<br>7 B<br>n<br>n |
| LD IX, × | | | | | | | D D<br>2 1<br>n<br>n | D D<br>2 A<br>n<br>n |
| LD IY, × | | | | | | | F D<br>2 1<br>n<br>n | F D<br>2 A<br>n<br>n |
| LD (nn), × | E D<br>4 3<br>n<br>n | E D<br>5 3<br>n<br>n | 2 2<br>n<br>n | E D<br>7 3<br>n<br>n | D D<br>2 2<br>n<br>n | F D<br>2 2<br>n<br>n | | |

**表4.2 16ビットロード命令変換表の一部**

## Challenge

16 ビットロード命令は 16 ビットのデータを扱うものですが，メモリとの間でデータをやり取りするときは，上位と下位の順番が逆になります．この点は重要ですからここではこれに気を付けながらプログラムを作っていきます．

**テーマ** メモリのD800H番地に33H，D801H番地に29Hを入れる

これは 16 ビットレジスタに値を入れて，これを一度にメモリに書き込んでしまえばよいことはわかりますね．ここではＨＬレジスタを使ってみましょう(**リスト 4.3**)．

### リスト 4.3

```
LD     HL,2933H      ……………Hレジスタに29H,Lレジスタに33Hがはいる
LD     (0D800H),HL  ……………D800H番地にLレジスタの内容(33H)がはい
                             り,D801H番地にHレジスタの内容(29H)が
                             はいる
RET ……………モニタに戻る
```

これは 16 ビットロード命令の変換表を 2 回見ればハンドアセンブルできます．アセンブルすると**リスト 4.3 A** のようになります．

## リスト 4.3A

```
D000 213329   LD    HL,2933H
D003 2200D8   LD    (0D800H),HL
D006 C9       RET
```

16ビットのデータは上位，下位が逆になることに注意

これは，前に出てきた８ビットのデータをメモリにロードする場合とまったく同じ形をしています．注意しなければならないのは，HL レジスタにロードする値が２９３３Hというように，メモリに入れたい順番と逆にするという点です．これはなぜかわかりますね．その下の〝LD　(0D800H), HL〞というロード命令ではLレジスタ，Hレジスタという順番でメモリにロードされるからです．

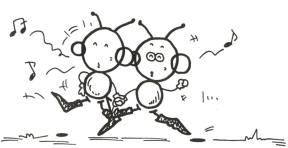
これを入力，実行して，Ｄ８００H番地にはいった結果を見たのが図4.9です．Ｄ８００H番地に３３H，Ｄ８０１H番地に２９Hがはいっているのがわかります．

```
>SD000
D000 FF-21 FF-33 FF-29 FF-22
D004 FF-00 FF-D8 FF-C9 FF-.      } プログラムの入力
>DD000 D007
D000 21 33 29 22 00 D8 C9 FF   !3)".リ/■ ……………プログラムの確認

>GD000 ……………実行

>DD800 D807
D800 33 29 FF FF FF FF FF FF   3)■■■■■■
         └─順番に注意
>■
```

図4.9 リスト4.3の実行例



# 5章

# ロード命令の応用と増減命令

# 5.1
# インデックスレジスタの使い方

前の章ではマシン語の命令の最初の例として，8ビットと16ビットのロード命令を見てみました．その中で出てきたレジスタは次の7種類です．

A，B，C，D，E，H，L

Aレジスタはアキュムレータ，そのほかの6つは汎用レジスタと呼ばれるものでした．これらはすべて8ビットですが，汎用レジスタの6つはそれぞれ3つのペアにして16ビットレジスタにもなるというものです．

Z80にはこのほかにもいくつかレジスタがありますが，ここでインデックスレジスタの使い方について説明します．

またこの後の節では，レジスタの値を1増やしたり，1減らしたりする命令を見てみます．レジスタ上で演算をするという命令は，この後，8章と9章で説明しますが，ここで説明する2つの命令(INCとDEC)は非常に簡単にレジスタの値を増減させることができるので，この章で使ってみようというものです．

## インデックスレジスタとは何か？

4.2 **8ビットロード命令**でみたように，HLレジスタを使って，メモリのアドレスを指定する方法があります．たとえば次のようにすると，HLレジスタの内容が9000Hであれば9000H番地の内容をDレジスタにロードします．

LD　D，(HL)

これと同じように，アドレスを指定してメモリの内容にアクセスするための16ビットレジスタがあります．それがIX，IYという〝インデックスレジスタ〟なのです．これ

は HL レジスタと違って，8ビットずつに分けることはできない 16 ビット専用のレジスタで，アドレスを指定するのが専門のレジスタなのです．

このレジスタにデータ(アクセスしたいメモリのアドレス)をロードするときは，次のようにふつうの 16 ビットロード命令と同じように書けます．

```
LD    IX,0B000H
```

この命令を実行すると，まず IX レジスタにB000Hという値がはいるので，メモリのB000H番地にアクセスしてその値をBレジスタに取ってきたい場合，次のようにします．

```
LD    B,(IX)
```

これだけなら HL レジスタと何ら変わらないのですが，インデックスレジスタにはもう少し便利な機能があります．それは**ディスプレースメント**というものです．これは IX レジスタで指したアドレスから，ある距離だけ離れたアドレスをアクセスするときに，その距離を指定するものです．ディスプレースメントは下のように，IX レジスタに定数を足すという形でニーモニックに書きます．

```
LD    B,(IX+5)
```

IX レジスタにB000Hがはいっているときに上の命令を実行すると，BレジスタにB005H番地の値をロードします．

ディスプレースメントに使える定数は1バイト，つまり8ビットです．ただしこれは8ビットの符号付きの定数（2の補数表示）なので，0から 255 までではなくて，**図5.1** に示すように－128 から＋127 までの数になります．

この図からわかるように，ディスプレースメントの値を変えることによってアクセスできるメモリの範囲は 256 バイトです．それよりも前や後のアドレスをアクセスしたい場合は，インデックスレジスタの内容を変えなければなりません．またインデックスレジスタそのものが指すアドレス(**図5.1** ではB000H番地)は正確には(IX)ではなくて，(IX＋0)になります．ですからさっきの例は正しくは

```
LD    B,(IX+0)
```

になります．

IY レジスタの使い方は IX とまったく同様ですから，ここでは説明を省略します．

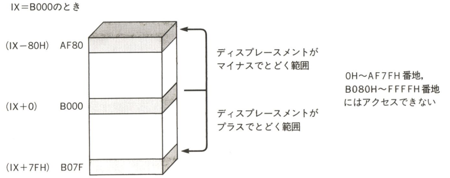



図5.1 ディスプレースメントでとどく範囲

## マシンコードの求め方

インデックスレジスタを使った命令をマシンコードに変換すると，必ず最初に１バイトのコードが付きます．IX,IY レジスタを使ったときは，それぞれ最初にＤＤＨ，ＦＤＨというコードが付くのです．たとえば，これらインデックスレジスタに 16 ビットのデータをロードするときは次のようにします．

```
LD    IX, 9000H…………DD  21  00  90
LD    IY, B092H…………FD  21  92  B0
```

２バイト目の２１Ｈというのがレジスタに 16 ビットの値をロードするということを表しています(これは〝ＬＤ　ＨＬ，xxxxH″の OP コードと同じ)．そして，その前にインデックスレジスタを使っているという意味でＤＤＨ，ＦＤＨを付けます．オペランドの部分は，これまでに出てきたのと同じように下位，上位の順に書きます．

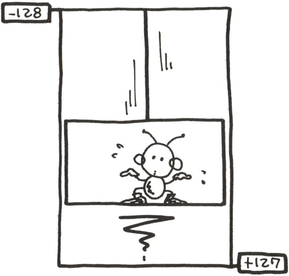
インデックスレジスタの内容が決まったら，次にそのアドレスのメモリにアクセスするわけですが，今度は前にＤＤＨ，ＦＤＨが付く上に，最後にディスプレースメントの１バイトが付きます．

```
LD   A,(IX+6)……………DD 7E 06
LD   (IY−1),43H………FD 36 FF 43
```

まず上の命令を見てください．7EHがAレジスタにメモリ内容を入れるという意味で，最後の06Hがディスプレースメントを表しています．下は，2バイト目と4バイト目の〝36，43〟が，43Hという定数をメモリに入れるという命令(36がOPコードになる)です．3バイト目のFFHは−1を表すディスプレースメントです．

インデックスレジスタはロードのほかにも演算命令などにも使われるので，それぞれの命令の変換表に載っています．これら変換表の中には3バイト目に〝d〟という文字がはいっていますが，これがディスプレースメントを表しています．インデックスレジスタでメモリにアクセスする場合には，必ずディスプレースメントを指定します．

インデックスレジスタを使うと必ず1バイト目がDDHかFDHになります．またメモリをアクセスする場合には，さらに最後にディスプレースメントの1バイトが必要なので，命令がほかのものより長くなりますが，インデックスレジスタを使うと非常に便利な場合もあります．次のChallengeに出てくる配列を使ったりする場合がそうです．



## Challenge

▶ディスプレースメントの使い方

さっそくここではこのインデックスレジスタを使ってみたいのですが，今までのようにロード命令ばかりではおもしろくありません．そこでレジスタの値を1増やしたり減らしたりする INC, DEC という命令を使います．これらについては，詳しくはこの次で説明します．コードの求め方も次に出てきますから，とりあえずここではこのまま入力してください．

**テーマ**　D800H番地の値をX，D801H番地の値をYとする
X+1をD802H番地に，Y−1をD803H番地に入れる

### リスト 5.1

```
LD      IX,0D800H  ……………IXレジスタにアドレス(D800H)をロードする
LD      A,(IX+0)   ……………D800H番地のメモリの内容をAレジスタにロードする
INC     A          ……………Aレジスタの内容を1増やす
LD      (IX+2),A   ……………Aレジスタの内容をD802H番地にロードする
LD      A,(IX+1)   ……………D801H番地の内容をAレジスタにロードする
DEC     A          ……………Aレジスタの内容を1減らす
LD      (IX+3),A   ……………Aレジスタの内容をD803H番地にロードする
RET
```

最初に IX レジスタにＤ８００Ｈを入れていますから，ディスプレースメント0から3までは，それぞれＤ８００Ｈ番地からＤ８０３Ｈ番地になります．ＸとＹのアドレスはそれぞれ(IX+0)，(IX+1)に相当します．1増やしたり減らしたりした結果を入れるアドレスも IX レジスタのディスプレースメントで示しています．

**リスト 5.1** をアセンブルしたのが**リスト 5.1 A** です．インデックスレジスタに関する命令は，どれも最初にＤＤＨ(IY のときはＦＤＨ)が付き，メモリを参照するときには，ディスプレースメントの1バイトが付きます．

## リスト 5.1A

```
      ……IXレジスタを使った命令にはDDHが付く
D000 DD2100D8          LD    IX,0D800H
D004 DD7E00 …          LD    A,(IX+0)
D007 3C                INC   A
D008 DD7702 … ディスプレースメントの値  LD    (IX+2),A
D00B DD7E01 …          LD    A,(IX+1)
D00E 3D                DEC   A
D00F DD7703 …          LD    (IX+3),A
D012 C9                RET
```

このプログラムはこれまでどおりモニタで入力しますが，実行の前に準備が必要です．それは，最初にＤ８００Ｈ番地とＤ８０１Ｈ番地にＸとＹの値をセットしておくことです．図5.2のように５と３を入れてみてください．

それが終わったら実行してみましょう．実行の仕方はこれまでどおり〝ＧＤ０００〟です．暴走しなければすぐにモニタに戻ってきますから，結果を確かめてみます．うまくいけばＤ８０２Ｈ番地に〝０６〟，Ｄ８０３Ｈに〝０２〟がはいっているはずです．

```
>SD000 ↵
D000 FF-DD FF-21 FF-00 FF-D8
D004 FF-DD FF-7E FF-00 FF-3C       プログラムの入力
D008 FF-DD FF-77 FF-02 FF-DD
D00C FF-7E FF-01 FF-3D FF-DD
D010 FF-77 FF-03 FF-C9 FF-.
>DD000 D013 ↵
D000 DD 21 00 D8 DD 7E 00 3C  ン!.リン~.く
D008 DD 77 02 DD 7E 01 3D DD  ンw.ン~.=ン   プログラムの確認
D010 77 03 C9 FF              w./■

>SD800 ↵  ↓        ↓  ――Xとの値をセットする
D800 33-05 29-03 FF-.
>GD000 ↵ ……………実行

>DD800 D803 ↵ ……………実行結果の確認
D800 05 03 06 02                 ....
      X   Y  X+1 Y-1
>■
```

図5.2 リスト5.1の実行例

▶配列への応用

これまでにまだロード命令しか説明していないので，あまりおもしろいプログラムはできません．そこで，ここではどういうところでこのインデックスレジスタを使えばよいかということを考えてみましょう．

**リスト**5.1のプログラムは，2つのメモリの内容に1を足したり引いたりした値を，別のメモリに入れました．プログラムの中ではメモリは変数の値を入れる場所として使われますから，これはちょうど次のようなことをしたのと同じです．X，Y，A，Bの4つの変数がＤ８００Ｈ番地からの4バイトにあたります．

A=X+1　：　ＡはＤ８０２Ｈ番地のメモリ
B=Y−1　：　ＢはＤ８０３Ｈ番地のメモリ

ところでこれだけのことをするのならば，別にインデックスレジスタを使う必要はありません．メモリのアドレス指定はHLレジスタを使うこともできますし，Aレジスタにロードするのならば直接アドレスを指定することもできます．逆にインデックスレジスタを使うとマシンコードが長くなるので不利にも思えます．

もちろんこのように計算するのが1回や2回ならインデックスレジスタは使うこともないのですが，これが連続して何回も行うとなると，ガゼン有利になります．これがインデックスレジスタの配列への応用です．

たとえば上の4つの変数がすべて0から99までの配列であったとしましょう．そして同じように2つの値の和と差を計算するということを100回繰り返すとすると，BASICプログラムでは次のようになります．

```
FOR  I=0  TO  99
  A (I)=X (I)+1
  B (I)=Y (I)-1
NEXT
```

BASICの場合，上のプログラムを実行するには，その前に〝DIM　A(99)″というように配列の大きさを宣言します．するとこの時点で4つの配列は，メモリの中で**図**5.3のように作られます．

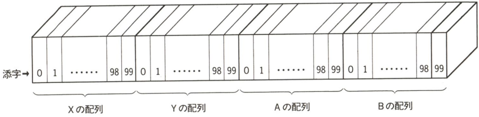



図5.3 配列のとり方 その1

さてこのような配列をとったとき，マシン語ではどう処理するでしょうか？ たとえばIXレジスタでX(0)のアドレスを指しておきますと，Y(0)は100バイト先ですから，(IX+100)になります．しかし次のA(0)になりますと，200バイト先になりますから，もうディスプレースメント(−128〜+127)で届く範囲から出てしまいます．

そこで4つの配列を，**図5.4**のように添字ごとにまとめておきます．こうすると2つの代入文は，**リスト5.1**のようにすれば，IXレジスタの内容を1つ決めるとX,Y,A,Bという4つの配列にアクセスすることが可能になります．実際には，この操作をIXを4増やしながら(これは添字を1進めることに相当する)100回繰り返せばよいのです．

```
FOR I=0 TO 99 (100回の繰り返し)
    リスト5.1のプログラム(RETは除く)
    IXを4増やす
NEXT
```

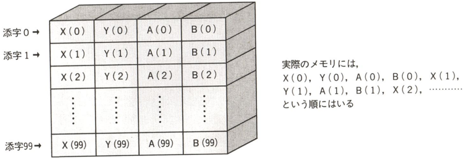



図5.4 配列のとり方 その2

# 5.2

# レジスタの値を増減する命令

ここで紹介する命令はINC（インクリメント）命令とDEC（デクリメント）命令の2種類だけです．これらはオペランドにレジスタをとって，その内容を1増やしたり減らしたりするものです．またHLレジスタやインデックスレジスタでメモリの内容を指定することもできます．

これらのニーモニックの形は次のとおりです．

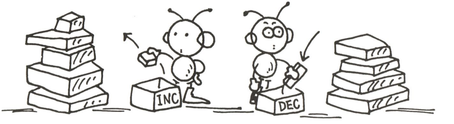
① INC　B
② DEC　(HL)

INCというのは〝インクリメント〟といってレジスタやメモリの内容を1増やし，DECは〝デクリメント〟といって1減らします．①はBレジスタの内容を1増やし，②はHLレジスタでアドレスを指定されるメモリの内容を1減らす命令です．たとえばHLレジスタに9000Hがはいっていれば，9000H番地の内容から1を引くのです．

このINC命令とDEC命令は8ビットレジスタばかりでなく，HLやIX, SPなどの16ビットレジスタに対して行うことも可能です．

たとえば③のようにすると，HLレジスタの内容を1増やします．この場合LレジスタがFFHから1増えた場合には，上位のHレジスタが1増えます．

③ INC　HL



## Challenge

INC／DEC 命令を使ってプログラムを作ってみましょう．といっても，実はさきほどインデックスレジスタの Challenge のところで作ってみました．これはAレジスタ以外のレジスタやメモリの内容に対しても演算が可能というところが特徴です(8章参照)．

その特徴を活かして HL レジスタに応用すれば，アドレスの指定を1つずつずらしていくことができます．これによって連続したメモリにアクセスすることができます．

**テーマ**　D800H番地から順番に4，3，2，1という値を入れる

このテーマは，アドレスは1つずつ増やし，データは1つずつ減らしてロードしていくという手法で実現できます．ここではBレジスタ(データ)と HL レジスタ(アドレス)を使ってみましょう(**リスト 5.2**)．

### リスト 5.2

```
D000 0604      LD    B,4        ……… Bレジスタにメモリに入れる値(4)をロードする
D002 2100D8    LD    HL,0D800H  …… HLレジスタにアドレス(D800H)をロードする
D005 70        LD    (HL),B     …… D800H番地に4をロードする
D006 05        DEC   B          …… Bレジスタの内容を1減らす
D007 23        INC   HL         …… HLレジスタの内容を1増やす
D008 70        LD    (HL),B     …… D801H番地に3をロードする
D009 05        DEC   B          …… Bレジスタの内容を1減らす
D00A 23        INC   HL         …… HLレジスタの内容を1増やす
D00B 70        LD    (HL),B     …… D802H番地に2をロードする
D00C 05        DEC   B          …… Bレジスタの内容を1減らす
D00D 23        INC   HL         …… HLレジスタの内容を1増やす
D00E 70        LD    (HL),B     …… D803H番地に1をロードする
D00F C9        RET
```

**リスト 5.2** を実行して，D800H番地以降をダンプしてみると，**図 5.5** のように，順番に4，3，2，1という数がはいっているはずです．

```
>SD000 ⏎
D000 FF-06 FF-04 FF-21 FF-00 ┐
D004 FF-D8 FF-70 FF-05 FF-23 │ プログラム入力
D008 FF-70 FF-05 FF-23 FF-70 │
D00C FF-05 FF-23 FF-70 FF-C9 ┘
D010 FF-.
>DD000 D00F ⏎
D000 06 04 21 00 D8 70 05 23   ..!.ﾘp.#  } プログラムの確認
D008 70 05 23 70 05 23 70 C9   p.#p.#pﾉ  }

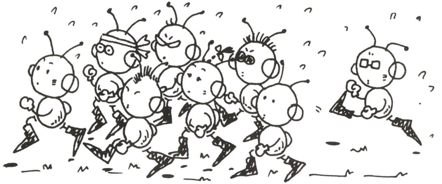
>GD000 ⏎……………実行

>DD800 D803 ⏎
D800 04 03 02 01 ……………結果の確認   ....

>■
```

図5.5 リスト5.2の実行例

このプログラムを見ると，同じ命令のパターンが繰り返されています．はっきりいってこれはウマいプログラムの作り方ではありません．本当はFOR～NEXTのようなループを作る命令が欲しいところです．これは11章のところで出てきますから，ここではこれで我慢してください．



# 6章

# サブルーチンコールとBIOS

# 6.1
# サブルーチンの果たす役割

BASICの中には，GOSUB／RETURNという命令があって，これでサブルーチンを呼び出すことができます．たとえば**リスト6.1**では，150行目に〝GOSUB　1500″という命令がありますが，これが実行されると，1500行から始まるサブルーチンを実行します．そして1590行目のRETURN文に会うと，サブルーチンが終わって，これを呼び出したGOSUB文の次に戻ります．

このように，ある仕事をサブルーチンにしておくと，それを1つの命令で呼び出すことができます．特にプログラムの中で，同じ仕事を何度も実行したいという場合には，その部分をサブルーチンにしておくと非常に便利ですし，プログラム自体も読みやすくなります．

**リスト 6.1**

```
100 REM SAMPLE PROGRAM
 :
150 GOSUB 1500 ・・・・・・・・・・・・1500行からのサブルーチンを呼び出す
 :
1500 REM ｶﾞﾒﾝ ｾｯﾃｲ
1510 SCREEN 2 : CLS
1520 LINE (0,0)-(255,191),15,B
1530 LINE (100,0)-(100,191),15     } サブルーチン
 :
1590 RETURN
```

マシン語の場合も同じで，GOSUB文と同じような働きをするCALL(コール)という命令があります．またRETURN文に相当するRET(リターン)命令があります．この2つを使えばマシン語でもサブルーチンを呼び出すことが可能になります．

## マシン語のサブルーチン

これまでのChallengeでは，いくつかサンプルプログラムを作ってみましたが，どれもみなプログラムの最後にRET命令が付いていましたね．これがこの節で説明するリターン命令なのですが，これが付いているということはサンプルプログラム自体はサブルーチンであったということになります．つまり，マシン語のプログラムをBASICモニタからサブルーチンコール(サブルーチンの呼び出しを〝コール〟という)していたわけです．

モニタのGコマンドは，BASICからマシン語プログラムのサブルーチンコールを行います．ですからそのコールされたマシン語プログラムにRET命令があれば，そこでBASICに戻ってきて，再びコマンド待ちの状態になるのです．

サブルーチンの形になっているマシン語プログラムは，他のマシン語のプログラムからコールすることもできます．同じマシン語のサブルーチンをコールするのに，マシン語からならCALL命令を使いますし，BASICからならUSR関数(15章参照)というものを使うわけです．

マシン語でプログラムを作るときは，プログラムすべてをサブルーチンの形にしておきます．ふつうサブルーチンがあれば，それをコールするメインルーチンというものがあるのですが，マシン語の場合にはメインルーチンにもRET命令を付けてサブルーチンの形にします．

## BIOSについて

この節でCALL命令を説明するのにはもう1つ目的があります．それはBIOS(Basic Input Output System)というものについて解説するためです．

BASICでは文字をプリントしたり，音を出したりすることができます．これは，BASICインタープリタの中にそういう働きをするサブルーチンがあるからなのです．これらのサブルーチンの集まりをBIOSと呼び，これを利用すると画面に文字を出したり，キーボードやジョイスティックの状態を調べたりと，様々なことが簡単にできるようになります．

# 6.2

# サブルーチンの形式

マシン語でサブルーチンを作るときは，図6.1のようにプログラムを書いて，その最後にRET命令を置きます．これをハンドアセンブルして，メモリの任意のアドレスから入れていきます．この過程は，これまでのChallengeでプログラムを作ったときと同じです．

```
このサブルーチンの → D000H     LD    A,(0D800H) ）
先頭アドレス          :         :         :      ｝サブルーチンの中では，1つ
                      :         :         :      ｝のまとまった仕事を処理でき
                      :         LD    (HL),B     ）るような形にしておく
                    D018H     RET
                              └サブルーチンの最後にはRET命令を入れる
```

**図6.1 サブルーチンの形**

このプログラムを別のアドレスからサブルーチンとしてコールするときは，そのサブルーチンの先頭アドレスを指定します．つまり，図6.1のサブルーチンをコールするときは，ニーモニックは次のようになります．

①CALL　0D000H

このように，プログラムのはいっているアドレスがオペランドに置かれます．これはちょうどBASICのGOSUB文で行番号を指定するのと同じです．RETURN文にあたるものは，前に出てきたとおり，

②RET

です．これはオペランドを付けません．

## ラベルについて

さてCALL命令を使ってプログラムを作る場合，1つ困ったことがあります．それはオペランドにプログラム(サブルーチン)のアドレスを指定しなければならないことです．ニーモニックでプログラムを書いているときは，まだサブルーチンがどのアドレスから始まるかというのがわかりません．なぜならある命令の前までアセンブルがすんでいないと，そのサブルーチンのアドレスがわからないからです．

BASICの場合ならばサブルーチンは1000行目から作るというつもりで，メインルーチンの方ではひとまず〝GOSUB 1000〟などと書くことができます．しかしマシン語の場合このように最初にサブルーチンのアドレスを決めてしまうと，後でアセンブルしたときメインルーチンとサブルーチンの間が大きく開いてしまったり，メインルーチンとサブルーチンの2つが重なってしまったりという不都合が生じます．

そこでマシン語でプログラムを作る場合には，サブルーチン(メインルーチンもモニタから呼ばれるのでサブルーチンと考えてよい)に，**リスト6.2**のように，ニーモニックでプログラムを作る段階で名前を付けます．

### リスト 6.2

```
KEISAN: LD      A,(HL) ············サブルーチンの先頭
:······ラベル
        ADD     A,8
        INC     HL
        LD      (HL),A
          :
        RET  ·············サブルーチンの終わり．このサブルーチンをコールし
                          たメインルーチンに戻る
```

このようにルーチンに付ける名前のことを**ラベル**といいます．ラベルはふつう英文字で始まる文字列で付けます．2文字目以降には英文字も数字も使えます．ただし〝LD〟とか〝CALL〟というように，ニーモニックと同じ言葉をラベルにしてはいけません．ラベルの後ろには〝:〟(コロン)を付けて，ニーモニックとの区別を明確にしておきます．

このようにラベルを付けたサブルーチンをコールするときのニーモニックは

③CALL　KEISAN

のように，オペランドにラベルをそのまま書けばよいのです．

このようにサブルーチンを使うときはそれにラベルを付けて，それをコールする命令の方でもオペランドにアドレスを書かず，そのラベルをコールするようにします．こうすればプログラムをニーモニックで書いている段階では，それぞれのサブルーチンがメモリのどのアドレスにはいるかということをまったく意識する必要がないからです．

## マシンコードの求め方

ここで出てきたCALL命令とRET命令のマシンコードは，それぞれ以下のとおりです．

```
CALL  0D218H ……  CD  18  D2
RET ……………………………  C9
```

まずRET命令はオペランドなしですからC 9 Hの1バイトだけです．CALL命令はOPコードとしてCDHがあり，その後の2バイトでコールするアドレスを指定します．アドレスは例によって16ビットのデータですから，上のように下位，上位の順に入れます．

しかしCALL命令のところでラベルを使った場合は困りますね．この場合，そのラベルのサブルーチンがどこのアドレスから始まるかがわからないと，CALL命令のオペランドが書けません．このようにラベルを使ったときのオペランドの決め方は，Challengeのところで詳しく説明します．

さて**表6.1**でCALL命令，RET命令を見ると，ほかにも似たような命令がいくつかいっしょになっています．これらは条件コール命令，条件リターン命令というものです．

条件コール命令や条件リターン命令については，フラグを説明した後11章で説明します．ただ，いまの段階では，CALL命令やRET命令はどの変換表を見ればよいかということを覚えておいてください．

| × | 無条件 | キャリー | ノン キャリー | ゼロ | ノン ゼロ | パリティ 偶数 | パリティ 奇 数 | 負 | 正 |
|---|---|---|---|---|---|---|---|---|---|
| | | C | NC | Z | NZ | PE | PO | M | P |
| CALL ×,nn | CD n n | DC n n | D4 n n | CC n n | C4 n n | EC n n | E4 n n | FC n n | F4 n n |
| RET × | C9 | D8 | D0 | C8 | C0 | E8 | E0 | F8 | F0 |

表6.1 コール・リターン命令

## Challenge

**▶サブルーチンの作成**

ここではCALL／RET命令を使って，サブルーチンコールを行ってみます．この2つをBASICのGOSUB文とRETURN文のように使えば，サブルーチンコールは実現できるのですが，そのほかに次のような点に気を付けて読んでいってください．

- ○ ラベルがあるときのアセンブルのしかた
- ○ サブルーチンの上手な作り方

**テーマ** メモリのHLレジスタの指すアドレスから，順に1と2を入れるサブルーチンを作る

まずテーマにあげたサブルーチンから作っていきます．サブルーチンがコールされるときには，あらかじめHLレジスタには値(アドレス)がはいっているものとします．

### リスト 6.3

```
SBRTN1:  LD      (HL),1  ……………HLレジスタにはあらかじめ値がはいっている
         INC     HL      ……………HLレジスタを1増やす
         LD      (HL),2  ……………HLレジスタで指定されたアドレスに2をロードする
         RET ………メインルーチンへ戻る
```

このサブルーチンをコールする前には，1と2を入れるメモリのアドレスをHLレジスタに入れておかなければなりません．それではこのサブルーチンを2回コールするメインルーチンを書いてみましょう．

## リスト 6.4

```
MAIN:   LD      HL,0D800H ┈┈ HLレジスタにアドレスをロードしてから
        CALL    SBRTN1        SBRTN1をコールする
        LD      HL,0D810H ┈┈ もう一度HLレジスタに別のアドレスをロード
        CALL    SBRTN1        してからSBRTN1をコールする
        RET ┈┈┈┈┈┈┈┈┈モニタに戻る　パラメータ
```

**リスト6.4**では必要な情報(アドレス)をレジスタに入れてから，サブルーチンをコールしています．このようにサブルーチンをコールするとき設定する値を，**パラメータ**(引数)といいます．この場合，1，2という定数を入れるメモリのアドレスをパラメータとしてHLレジスタに入れます．するとサブルーチンの方ではそのアドレスのメモリに1と2という値を続けて入れてくれるのです．

このプログラムはまとめてアセンブルします．順番は後でこのプログラム全体をモニタから実行するときにわかりやすいようにするため，メインルーチンをサブルーチンの上にします．

**リスト6.4**と**リスト6.3**をつなげて，D000H番地から入れていくものとして最初からアセンブルしていくと，まずCALL命令のところで困ります．つまりこの時点では，サブルーチンSBRTN1というラベルが何番地になるのかがわからないのです．

こういう場合はそのオペランド部分は飛ばして続きをアセンブルします．CALL命

## リスト 6.5

```
D000 2100D8         MAIN:   LD     HL,0D800H  ┐
D003 CD□□┈┈オペランドのラベル CALL  SBRTN1     │ メインルーチン
         の部分は空けておく  LD     HL,0D810H  │ リスト6.4 の部分
                            CALL   SBRTN1     │
                            RET               ┘
                    SBRTN1: LD     (HL),1     ┐
                            INC    HL         │ サブルーチン
                            LD     (HL),2     │ リスト6.3 の部分
                            RET               ┘
```

令の OP コードのＣＤＨがまずＤ００３Ｈ番地にはいりますが，**リスト 6.5** のように，その次の２バイトのオペランド部分は空けておきます．そしてＤ００６Ｈ番地からロード命令をアセンブルするのです．２番目の CALL 命令でも同様です(**リスト 6.6**)．

## リスト 6.6

```
D000 2100D8      MAIN:    LD      HL,0D800H
D003 CD□□                 CALL    SBRTN1
D006 2110D8               LD      HL,0D810H
D009 CD□□                 CALL    SBRTN1
D00C C9                   RET
D00D             SBRTN1:  LD      (HL),1
                          INC     HL
                          LD      (HL),2
                          RET
```

そうするとメインルーチンの RET 命令が終わったところで，ラベル SBRTN1 のアドレスがＤ００ＤＨ番地であることがわかります．そこでさっき空けておいた CALL 命令のオペランドの部分に０ＤＨ，Ｄ０Ｈを入れるのです．CALL 命令のオペランドはアドレスですから，例によって下位，上位の順にします．

SBRTN1 の部分は楽にできます．サブルーチンの RET 命令まで行けばアセンブル完了です(**リスト 6.7**)．

## リスト 6.7

```
D000 2100D8      MAIN:    LD      HL,0D800H
D003 CD0DD0               CALL    SBRTN1
D006 2110D8               LD      HL,0D810H
D009 CD0DD0               CALL    SBRTN1
D00C C9                   RET
D00D 3601        SBRTN1:  LD      (HL),1
D00F 23                   INC     HL
D010 3602                 LD      (HL),2
D012 C9                   RET
```

このプログラムをモニタで入力，実行したのが**図 6.2** です．値を入れたメモリはＤ８００Ｈ～Ｄ８０１Ｈ番地とＤ８１０Ｈ～Ｄ８１１Ｈ番地ですから，この部分を確かめてみましょう．うまくいけばどちらにも，０１Ｈと０２Ｈが続いてはいっています．

```
>SD000 ↵
D000 FF-21 FF-00 FF-D8 FF-CD
D004 FF-0D FF-D0 FF-21 FF-10
D008 FF-D8 FF-CD FF-0D FF-D0      プログラムの入力
D00C FF-C9 FF-36 FF-01 FF-23
D010 FF-36 FF-02 FF-C9 FF-.
>DD000 D012 ↵
D000 21 00 D8 CD 0D D0 21 10   !.リヘ.ミ!.
D008 D8 CD 0D D0 C9 36 01 23   リヘ.ミノ6.#   プログラムの確認
D010 36 02 C9                  6.ノ

>GD000 ↵……………実行

>DD800 D801 ↵
D800 01 02
                 D800H番地とD810H番地からの2
>DD810 D811 ↵    バイトにそれぞれ01Hと02Hが書き   確認
D810 01 02       込まれている

>■
```

図6.2 リスト6.5の実行例

## NOP 命令の利用

さて，いまのメインルーチンはサブルーチンを2回コールしていますが，最初のCALL命令だけを残し，2回目の方を取ってしまいたいとします．

ハンドアセンブルからやり直すこともできますが，そうするとSBRTN1のラベルも変わってしまいます．そこでサブルーチンのアドレスを変更しないで，このまま（モニタにプログラムがはいった状態で）このCALL命令をなくしてしまう方法があります．それは図6.3のように，このCALL命令の3バイトのところに0 0 Hを書き込むという方法です．

0 0 Hというのは，実はNOP（ノップ）といって〝何もしない〟というオペランドなしの1バイト命令です．このように別の命令の場所にNOP命令を入れるとその命令を実行しないことができるので，デバッグのときなどにときどき使われます．

これは1バイト命令ですから，CALL命令のような3バイト命令をつぶそうと思ったら，NOP命令を3つ入れなければなりません．

```
>SD009
D009 CD-00 0D-00 D0-00 ……………2番目のCALL命令をつぶす
D00C C9-.
>DD000 D012
D000 21 00 D8 CD 0D D0 21 10   !.リヘ.ミ!.
D008 D8 00 00 00 C9 36 01 23   リ.../6.#
D010 36 02 C9  ↑              6./
               └─00Hに書き換える
```

図6.3 命令の削除

このプログラムを実行する前に，さっき０１Hと０２Hを書き込んだＤ８００H番地からとＤ８１０H番地からの２バイトに別の適当な値を書き込んでおいてください．実行すると図6.4のように，Ｄ８００H番地からの方にはまた０１Hと０２Hがはいりますが，Ｄ８１０H番地からの方はメモリの内容が変わっていません．

```
>SD800
D800 01-00 02-00 02-.    NOPを書き込んだ結果を確認するために，あらかじめ別の値を
>SD810                    書き込んでおく(ここでは00Hを書き込んでいる)
D810 01-00 02-00 FF-.
>GD000 ………実行

>DD800 D801
D800 01 02 ……………はいっている        最初のCALL命令は実行され
                                      ているが，2番目のCALL命令
>DD810 D811                           はNOP(00H)で埋めてしまっ
D810 00 00 ……………はいっていない      たのでD810H，D811H番地の
                                      内容は書き換えられていない
>■
```

図6.4 NOP命令の確認

このようにNOPという命令は，一時的にある命令を飛ばして実行するときに使うと非常に便利です．いまのCALL命令のように3バイトの命令を埋めるときは，3つのNOP命令を入れます．これを最初のCALL命令のOPコード(ＣＤH)のメモリだけNOPにしたりすると，その後にあるオペランドの部分(０ＤH，Ｄ０H)を次の命令と思って実行してしまいます．こうなると，まず間違いなく暴走しますから，NOPを使うときは，必ず命令をすべて(OPコード＋オペランド)消すようにしてください．

# 6.3

# CALL命令のメカニズム

ここで，マシン語のサブルーチンコールはどのように行われるかということを説明しましょう．図6.5はサブルーチンコールをしているマシン語プログラムの例です．

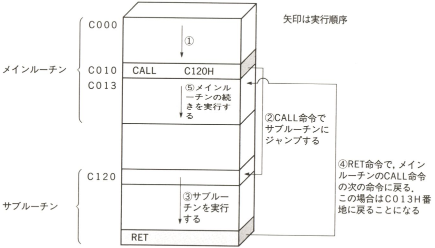



**図6.5 メモリにはいったサブルーチンとCALL命令**

図6.5ではメインルーチンがＣ０００Ｈ番地から始まっています．CPUはこのアドレスから命令を読み出しては実行していきますが，Ｃ０１０Ｈ番地で〝CALL　Ｃ１２０Ｈ″という命令に出会ったとします．するとCPUは，その次のＣ０１３Ｈ番地(CALL命令は3バイト)の命令ではなく，Ｃ１２０Ｈ番地からのサブルーチンへジャンプします．このサブルーチンの最後にはRET命令があり，これが実行されるとこのサブルーチンをコールしたCALL命令の次，すなわちＣ０１３Ｈに戻ります．これで

Ｃ０００Ｈから始まるメインルーチンから，Ｃ１２０Ｈのサブルーチンをコールすることができたわけです．

いまの例では，Ｃ０１０Ｈ番地からＣ１２０Ｈ番地にジャンプしたのはそんなに不思議ではありません．ジャンプ先のＣ１２０Ｈ番地はプログラムの中(オペランド)に書かれているので，これを PC に入れれば次はそのアドレスから命令を読み出して実行してくれます．

不思議なのはサブルーチン最後の RET 命令の後，Ｃ０１３Ｈに戻れることです．RET 命令にはアドレスを指定するオペランドがありません．というよりサブルーチンはどこからコールされるかわかりませんし，プログラムのどのアドレスからも同じサブルーチンがコールされる可能性があります．ですからプログラムを作る時点ではこのリターンアドレスはわかりません．

## スタック

このリターンアドレスはそのサブルーチンがコールされたときに決まり，どこかに覚えておかなければならないのです．それを覚えておくところが**スタック**というところなのです．スタックはメモリの中にとられ，そのスタックのアドレスを指定するのが SP(スタック・ポインタ)というレジスタなのです．

スタックに値を出し入れするときは必ず２バイト単位で行われます．そして，出し入れするごとに SP の値は２ずつ増えたり，減ったりします．

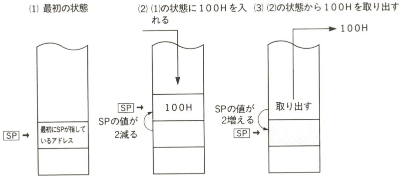



図6.6 スタックとSPの働き

図6.6はスタックとSPの働きを示したものです．SPは(1)のようにスタックのあるアドレスを指していますが，この上に１００Hという数を入れると，SPは2減っていま入れた１００Hのアドレスを指すようになります．ここから１００Hを取り出すと，SPはまた2増えて，もとの状態に戻ります．

サブルーチンコールには，このスタックが使われているのです．図6.7の例でいうと，Ｃ０１０H番地にあるCALL命令を読み出した時点(まだ命令は実行されていない)で，PCはＣ０１３Hになっています．ですからＣ１２０H番地にジャンプする前に，このPCの値をスタックに入れておけばよいのです．すると，さきほどの１００Hと同じように，SPはいま入れたPC(リターンアドレスになる)の内容のはいったアドレスを指しています．それが終わった時点で，PCにＣ１２０Hを入れてサブルーチンにジャンプします．

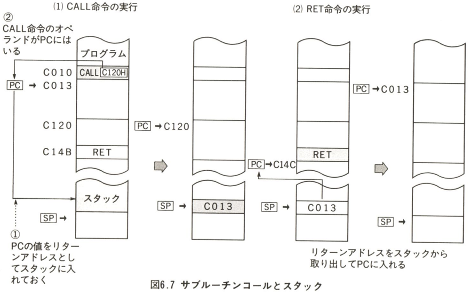



図6.7 サブルーチンコールとスタック

サブルーチンの中でRET命令に出会うと，これと逆の操作をします．つまりスタックのSPが指しているアドレスのメモリから2バイトを取り出してPCに入れ，SPを2増やします．PCには，さきほど入れたＣ０１３Hがはいりますから，これでリターンアドレスにジャンプすることができたわけです．

この原理を応用すれば，サブルーチンは何重にでも入れ子*にすることができます．図6.8がその例です．C０２０H番地でC１００H番地のサブルーチンをコールし，そのルーチンの中で，またC１８８H番地をコールしています．最初のCALL命令のときには，スタックには戻り番地C０２３Hがはいります．２回目のCALL命令のときには，スタックには，C１０BHがはいるわけですが，前のリターンアドレスは消えずに，その上に積み上げられます．ですからC１８８H番地からのサブルーチンから戻ったとき，まだスタックにはメインルーチンのリターンアドレスC０２３Hが残っているわけです．

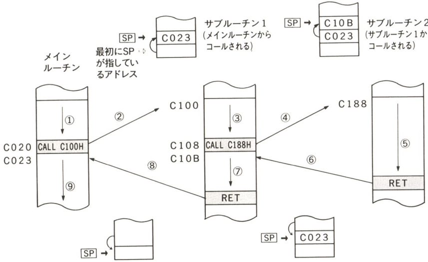



図6.8 サブルーチンの入れ子

このようにスタックの上にはどんどんリターンアドレスを積むことができますから，何重にでもサブルーチンをコールすることができます．このとき，サブルーチンの最初と最後(RET命令を実行する前)はSPの位置が同じであることに注意してください．これが違ってしまいますと，うまくもとのアドレスに戻れなくなってしまいます．

*サブルーチンの中からまた別のサブルーチンをコールすること

# 6.4

# BIOSを使う

MSXのROM(0H番地から7FFFH番地まで)にはBASICインタープリタが書き込まれています．そしてこのインタープリタの中には，ユーザーが作ったBASICのプログラムを解読する部分と，様々な仕事を実行するBIOSの部分とがあります．〝様々な仕事〞というのは，画面に文字を出したり，ジョイスティックやキーボードが押されているかというようなことを調べたりというように，ユーザーやCPU以外のLSIとの通信が主なものです．

インタープリタはプログラムを解読し，PRINT文があれば文字を表示するBIOSルーチン，PLAY文やSOUND文があれば音を出すBIOSルーチンをコールします．ですからマシン語のプログラムの中からでも，BIOSルーチンをコールすれば，音を出したり文字を画面に表示したりすることができるはずです．

それでは，実際にBIOSルーチンをコールしてみましょう．これはモニタから直接Gコマンドでコールすることができます．図6.9のように〝G00C0〞を実行してください．

```
>G00C0 ⏎……………ROMの中のBIOSルーチンをコール(実行)する
>■
```

図6.9 BIOSコールの実行例

どうなりましたか？ スピーカーがちゃんとつながったマシンならば，ビープ音が出たはずです．このBIOSルーチンはインタープリタがBASICのBEEP命令を実行するときにコールされるものですが，マシン語のプログラムの中で〝CALL　00C0H〞を実行すれば，ビープ音を出すことができます．

BIOS ルーチンを利用することにはいろいろなメリットがあります．まず1つはよけいなプログラムを書く手間が省けるということです．たとえばビープ音にしても，I/O ポートなどマシンの詳しい構造を調べれば音を出すプログラムを作ることができます．しかし，それと同じものが同じメモリの中にあり，CALL 命令1つでそれが使えるのならば，わざわざ作る必要はないではありませんか！ また，これと同じ理由で，プログラムが小さくなるというメリットもあります．何しろ CALL 命令1つならば3バイトですむのです．

## BIOS エントリー表

MSX は，BIOS ルーチンの**エントリーアドレス**というものが公表されています．これは〝どのレジスタにどういう値をセットして，何番地をサブルーチンコールすればこういう仕事をしてくれる〟というものです．〝エントリー〟というのは入口という意味です．

この本の付録には，この〝BIOS エントリー表〟がありますので，詳しい内容についてはこれを見てください．BIOS ルーチンの中にはいまのビープ音のように，ただコールすれば事足りるというものもありますが，ふつういろいろな情報をレジスタに入れる必要があります．これが**エントリーパラメータ**というものです．そしてサブルーチンコールの結果として，ある値をレジスタに入れて返すものが**リターンパラメータ**です．

たとえば**リスト 6.8** は，画面に文字を1つ出力する〝CHPUT〟というルーチンを使った例です．これはエントリーパラメータとして，A レジスタに出力する文字のキャラクタコードを入れてからコールします(**表 6.2**)．

**リスト 6.8**

```
LD      A,48H ·············Aレジスタにキャラクタコードの48H(大文字のH)をロードする
CALL    CHPUT ·············BIOSのCHPUTルーチンをコールする
RET
```

| アドレス | 名　前 | 機　　　能 |
| --- | --- | --- |
| 00A2 | CHPUT | 画面に指定した文字を表示する<br>●エントリーパラメータ：Aレジスタにキャラクタコードを入れる<br>●リターンパラメータ　：なし<br>●使用レジスタ　　　　：なし |

表6.2 CHPUTのBIOSエントリー表

リスト6.8を実行すると画面に〝H″の文字が表示されます．

また表6.3の〝CHGET″をコールすると，キーボードからの入力を待ち，入力されたキャラクタコードがAレジスタにはいります．このように，コールした結果ある値がはいるものが，リターンパラメータといわれるものです．

| アドレス | 名　前 | 機　　　能 |
| --- | --- | --- |
| 009F | CHGET | 入力があるまで待ち，入力したキャラクタコードを求める<br>●エントリーパラメータ：なし<br>●リターンパラメータ　：<br>　Aレジスタに入力したキャラクタコードがはいる<br>●使用レジスタ　　　　：AF |

表6.3 CHGETのBIOSエントリー表

もう1つ，BIOSエントリー表には使用レジスタというものがあります．これはそのルーチンをコールしたときに，値が書き換わってしまうレジスタのことです．ここに書いてないレジスタは，そのBIOSルーチンの中で使ったとしてもルーチンから戻ってくるときにはこのルーチンにはいってきたときの値に戻っています．ですから，あるレジスタに値を保存しておいたままBIOSルーチンをコールしても，メインルーチンでは，まったく気にせずにレジスタを使うことができるわけです．

いままでのマシン語入門書は，あまりBIOSのことについて書かれていませんでした．それは，CPUが同じであっても，BIOSというのはマシン(BASICインタープリタ)の違いによって，エントリーアドレスも使い方もまったく異なるからです．しかし，MSXは，どのマシンを使ってもBIOSのエントリーアドレスは同じです．ですからこの本では，BIOSを積極的に使っていこうと思います．

## Challenge

▶BIOSルーチンの利用

ここではBIOSルーチンを使っていろいろなプログラムを作ってみましょう．この本の付録の〝BIOSエントリー表〟というのを見ると，かなりの数があります．これらは〝PSG関係〟とか〝VDP関係〟といったように，関係のあるものがまとめられています．たとえば００４１Ｈから００８ＤＨまではVDPに関するものばかりです．

これらのBIOSルーチンは，後で画面や音を扱うときに必要に応じて説明しますから，いますぐに理解する必要はありません．

BIOSルーチンの中で，その働きがいちばんわかりやすいのは，入出力でしょう．このChallengeでは次の３つのルーチンを使ってみます．

| アドレス | 名前 | 機能 |
| --- | --- | --- |
| ００９ＦＨ | CHGET | 押されたキーのキャラクタコードをＡレジスタに入れる |
| ００Ａ２Ｈ | CHPUT | Ａレジスタの文字を画面に表示する |
| ００Ｃ６Ｈ | POSIT | カーソルを動かす |

**テーマ** 入出力のBIOSルーチンを利用したプログラムを作る

まず最初にCHPUTを使ってみます．これはＡレジスタに入れたキャラクタコードの文字を画面に表示するというものです．これで〝@〟という文字を出力したいときは，まず〝@〟のキャラクタコード*を調べます．これは４０Ｈですから，それをＡレジスタにロードして，００Ａ２Ｈ番地をコールするというプログラムを作ればよいのです(リスト6.9)．

### リスト 6.9

```
LD      A,40H  ……………〝@〟のキャラクタコード(40H)をAレジスタにロードする
CALL    CHPUT  ……………1文字出力ルーチン(00A2H番地)をコールする
RET
```

*付録のキャラクタコード表参照

これをアセンブルしたのが**リスト 6.9 A** です.

## リスト 6.9A

```
D000  3E40             LD      A,40H
D002  CDA200           CALL    CHPUT
D005  C9               RET
```

これをモニタから入力して実行してみましょう.

```
>SD000↵
D000  FF-3E  FF-40  FF-CD  FF-A2 ┐
D004  FF-00  FF-C9  FF-.         ┘ プログラムの入力
>DD000  D005↵…………確認
D000  3E  40  CD  A2  00  C9          >@ヘ「.ノ

>GD000↵…………実行
@ …………カーソルのあった位置に，Aレジスタにロードしたキャラクタコードの文字が表示される
>■
```

**図6.10 リスト6.9の実行例**

どうです？〝@〟が表示されましたね．1文字だけではつまらないのでいくつか文字を表示してみましょう．同じようにAレジスタにキャラクタコードを入れて，0 0 A 2 H番地を繰り返しサブルーチンコールすればよいのです(**リスト 6.10**).

## リスト 6.10

```
D000  3E40             LD      A,40H  …………@のキャラクタコード40H
                                               をAレジスタにロードする
D002  CDA200           CALL    CHPUT  …………Aレジスタの文字を画面に
                                               表示する（@が表示される）
D005  3E41             LD      A,41H
D007  CDA200           CALL    CHPUT
D00A  3E42             LD      A,42H
D00C  CDA200           CALL    CHPUT
D00F  C9               RET
```

このプログラムを実行すると，**図 6.11** に示すように〝@ＡＢ〟という3文字が表示されます．このようにCHPUTというルーチンは1文字ごとには改行しないので，続けてコールすれば連続して文字を表示します．ちなみにモニタのコマンド待ちの状態

に戻ってきたときに改行して次の行にプロンプトが出るのは，モニタのプログラムの中(1090 行目)で PRINT 文を実行しているからです．

```
                          前と同じプログラムなのでスペースキーを押して
>SD000 ⏎                  メモリの内容は書き換えない
D000 3E-__ 40-__ CD-__ A2-__  ┐
D004 00-__ C9-3E FF-41 FF-CD  │ プログラムの入力
D008 FF-A2 FF-00 FF-3E FF-42  │
D00C FF-CD FF-A2 FF-00 FF-C9  ┘
D010 FF-.
>DD000 D00F ⏎
D000 3E 40 CD A2 00 3E 41 CD   >@へ「.>Aへ  } プログラムの確認
D008 A2 00 3E 42 CD A2 00 C9   「.>Bへ「.ノ

>GD000 ⏎ ……………実行
@AB ……………実行結果
>■
```

図6.11 リスト6.10の実行例

次は CHGET です．このルーチンの説明には〝入力があるまで待ち，入力したキャラクタコードを求める〟と書いてあります．つまりキーボードから入力した文字のキャラクタコードを A レジスタに入れて返してくるというものです．BASIC でいうと，〝INPUT＄(1)〟という関数と同じ働きをします．これと CHPUT を組み合せると，リスト 6.11 のようにキーから入れた文字を画面に出力するというプログラムができます．

## リスト 6.11

```
D000 CD9F00     CALL   CHGET …キーボードから入力された文字のキャ
                              ラクタコードをAレジスタに入れる
D003 CDA200     CALL   CHPUT …Aレジスタのキャラクタコードに対応
                              する文字を画面に表示する
D006 C9         RET
```

リスト 6.11 を実行すると，最初は何も文字が表示されません．これはキー入力を待っている状態です．そこで何かキーを押すとその文字が画面に表示されて実行が終わります．

この CHGET のようなルーチンは，プログラムがマシンの外部からデータを受け取

るときに使います．このCHGETは1文字だけで，しかもキャラクタコードを返すというものなので，とてもBASICのINPUT文のようには使えません．しかし1文字入力があるまで待ってくれるという特徴があるので，特定のキー入力に対する処理ができます．

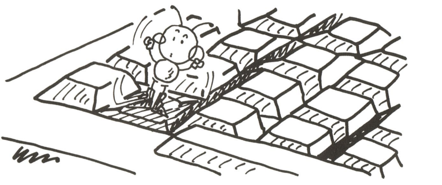
たとえばプログラムの最後で〝もう一度やりますか？〟というような質問を出し，それに対してユーザーはY(yes)かN(no)のキーで答えるとします．このプログラムでは，最後のところでメッセージを出してCHGETをコールします．

その結果AレジスタにはいってきたものがY(コードは59H)であればもう一度プログラムを最初から実行し，N(コードは4EH)であればプログラムの終了のアドレスにジャンプします．それ以外のキーが入力されたら再びCHGETをコールしているところに戻って入力をやり直します．

このようなプログラムは後で説明する比較命令とか，ジャンプ命令というような命令が必要になります．11章の条件ジャンプ命令のところでは，このCHGETを使ったプログラムを作ってみます．

### ▶カーソルを動かす

さて画面に文字を表示するCHPUTは，ただ単にAレジスタにセットされたキャラクタコードに対応する文字を，いまカーソルのある場所に表示するというものです．これはちょうどBASICのPRINT文と同じです．画面の指定した場所に表示したい場合には，その前にカーソルの位置を指定するLOCATE文のようなものが必要になります．それがPOSITというBIOSルーチンです．



LOCATE 文では少なくともカーソルの X 座標と Y 座標が必要です．POSIT はそれを H と L の 2 つのレジスタで指定します．モニタを使っている画面はスクリーンモード 0 ですから，H，L レジスタにはそれぞれ，**図 6.12** に示す範囲の数値を設定します．

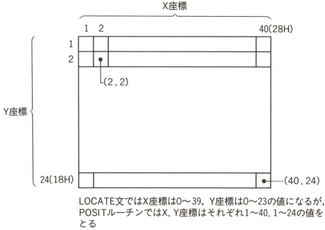



**図6.12 スクリーン 0 の画面の構造**

ここで注意したいのは，BASIC の LOCATE 文では X 座標も Y 座標も 0 から始まるのに対し，POSIT のエントリーパラメータは 1 から始まるということです．また，どちらも範囲を超えると，最大値(H レジスタは 2 8 H，L レジスタは 1 8 H)に設定されたものとみなされます．

この POSIT を使ってプログラムを作ったのが**リスト 6.12** です．BASIC の LOCATE 文と同じで，その後に文字を表示してみないと，どこにカーソルが移ったのかわからないので，CHPUT といっしょに使ってみます．

## リスト 6.12

```
D000 211008      LD    HL,0810H ……X座標8H，Y座標10H
D003 CDC600      CALL  POSIT
D006 3E42        LD    A,42H ………"B"のキャラクタコード
D008 CDA200      CALL  CHPUT
D00B C9          RET
```

図6.13が実行例ですが，このようにキャラクタ座標(7，15)の位置に"B"という文字が表示されています．BASICならば"LOCATE 7，15"とするところですが，POSITの指定はさきほどもいったように，両方1ずつ増やします．

```
>SD000 ⏎
D000 FF-21 FF-10 FF-08 FF-CD  ┐
D004 FF-C6 FF-00 FF-3E FF-42  │ プログラムの入力
D008 FF-CD FF-A2 FF-00 FF-C9  │
D00C FF-.                     ┘
>DD000 D00B ⏎
D000 21 10 08 CD C6 00 3E 42   !..ヘニ.>B  ┐ プログラムの確認
D008 CD A2 00 C9               ヘ｢.ノ      ┘

>GD000 ⏎ ……………実行

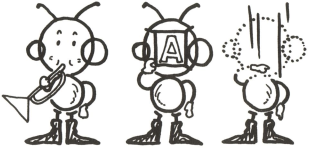


          B ……………これが出た場所が(08H，10H)の座標
>■
```

図6.13 リスト6.12実行例

▶その他おすすめのBIOS

ここで使ってみたBIOSはとりあえず3つだけです．ほかにもパラメータのいらないBEEP(ベルをならす)やCLS(画面をクリアする)などはコールするだけでそれぞれの機能が果せますから，ぜひ使ってみてください．最初にやってみたように，モニタから直接コールすることもできます．その他のルーチンは今後必要になった時点でそのつど説明します．



# 7章

# スタックとサブルーチンの利用

# 7.1
# サブルーチンを作るときの注意

この章ではスタックを利用する命令を紹介します．スタックは前の章にも出てきたように，サブルーチンをコールするときにリターンアドレスを入れておき，RET 命令の実行のときにもとのアドレスに戻るのに使われていました．しかしスタックにはリターンアドレスだけでなく，レジスタの内容を入れたり出したりすることもできます．それを行うのがこの章で見ていく PUSH（プッシュ）と POP（ポップ）という 2 つの命令なのです．

## 使用レジスタ

Z80 の 8 ビットレジスタはAからLまで 7 つしかありません．インデックスレジスタはアドレス指定が専門ですから，あまり計算などには使いません．ですから通常使えるレジスタというのは 7 つしかないといってもよいでしょう．プログラムを作るときには，これを BASIC の変数のようにいろいろなデータを入れておきますが，8 ビットレジスタが 7 つというのでは数が少なすぎます．

マシン語プログラムで〝何か〞をしようとすれば，すぐにレジスタには別の値がはいってしまいます．たとえばBレジスタの内容をＡ０２０Ｈ番地に入れたいとすると，Bレジスタは直接アドレスを指定してメモリにロードできませんから，次のようなプログラムを作ります．

```
①LD   A,B
  LD   (0A020H),A

②LD   HL,0A020H
  LD   (HL),B
```

このどちらかの方法でBレジスタの内容をＡ０２０Ｈ番地に入れることができますが，その代わりに①ではAレジスタ，②ではHLレジスタが使われてしまうため，これらにもともとはいっていた値は失われてしまいます．このように，レジスタを使用するために他の値がはいってしまうとことを，〝レジスタの内容が壊れる〟といいます．

しかしプログラムを作る際に，何か小さな仕事を１つさせるたびに，あるレジスタの内容が壊れてしまうというのでは，非常に不便です．

たとえば②のBレジスタをメモリに入れるサブルーチン(②の場合〝使用レジスタ〟はHL)を〝LDBM〟というラベルを付けて，別のルーチンからコールしたとします．ところがそのルーチンでHLレジスタを別のアドレスを指定するために使っている場合には，LDBMをコールする前にこのHLレジスタの内容を別のところに取っておかなければなりません．別のところといっても，レジスタの数はそんなにありませんから，空いているメモリに入れることになります．ですからコールする側では**リスト7.1**のようにして，HLレジスタを保存しなければなりません．

## リスト 7.1

```
LD      (0D800H),HL ……HLレジスタの内容をD800H番地に保存しておく
CALL    LDBM        ……サブルーチン"LDBM"をコールする
LD      HL,(0D800H) ……D800H番地に入れておいた値をHLレジスタに戻す
RET
```

上のようにLDBMをコールする前にHLレジスタの内容をメモリのＤ８００Ｈ番地に入れておいて，戻ってから同じアドレスのメモリからHLレジスタに取ってくれば，HLレジスタは保存されたことになります．しかし，これではサブルーチンをコールする前と後に3バイトずつのロード命令が必要になりますし，サブルーチンをコールするたびに，そのためのメモリの配置を考えてやらなければなりません．

今度は〝使用レジスタなし〟のBIOSルーチンCHPUTを使う場合を考えてみます．**リスト7.2**はCHPUTを2回コールして，BCレジスタにはいった2文字(16ビット)を画面に表示するプログラムです．



**リスト 7.2**

```
        LD      A,B
        CALL    CHPUT
        LD      A,C
        CALL    CHPUT
```

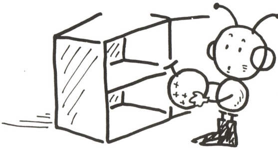
どうです，さっきよりもスッキリしていますね．1回目のCHPUTをコールしたときに，Cレジスタが使用されずに前の値が残っていますから，レジスタの値の保存のことなど考えずにBレジスタの内容をAレジスタに入れるだけでCHPUTをコールすることができるのです．

## サブルーチンでのレジスタの保存

いま説明したように，サブルーチンをコールしたときにレジスタの値が壊れない方が，レジスタの内容を保存する手間がないので便利です．このようなサブルーチンを作るには，レジスタの値をスタックに入れたり出したりするという方法を使います．この操作を行うのがPUSH命令とPOP命令なのです．

この節に出てくる命令はこの2つだけです．命令自身は難しくありません．むしろこれらをどんなときに，どのように使うかということがポイントです．

スタックを使う上で，SPの働きは非常に大切ですから，しっかりと頭に入れておいてください．スタックの扱いを誤ると，サブルーチンの動作が正しく行われません．また，PUSH命令1つが原因で暴走してしまうこともよくありますから，この点は気を付ける必要があります．



# 7.2

# PUSH／POP命令とスタック

前の章ではCALL／RET命令の働きとともにスタックの構造について説明しました．ここではユーザーの立場から，PUSH／POP命令を使ってどのようにスタックを使うかについて見ていくことにします．

## スタックの構造

スタックはメモリの中に取られ，そのアドレスをSPが指しています．このSPが指しているところを〝スタックトップ〟といいます．スタックに値を入れると，スタックトップは1つ(2バイト)上がります．取り出すときはスタックトップから取り出して，スタックトップは1つ下がります．メモリの中のスタックは，アドレスの若い方に向かって伸びますから，上がるといえばアドレスの小さい方に，下がるといえばアドレスの大きい方にSPが移動するのです．

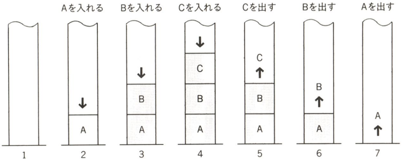



図7.1 FILOのしくみ

スタックを使うときにはデータの出し入れの順序が大切です．たとえば図7.1のように，〝A，B，C〟という順に3つのデータを入れたときは，〝C，B，A〟という順序でしか取り出すことはできません．このようなデータ格納形式を，ファーストイン・ラストアウト(First In Last Out)といいます．最初に入れた値が最後に取り出されるという意味です．

スタックにレジスタの内容を入れる命令がPUSHで，逆にスタックから取り出した値をレジスタに入れるのがPOPです．図7.1ではA，B，Cという文字を入れていますが，実際のスタックには16ビットの数値がはいります．サブルーチンをコールするときにはリターンアドレスがはいりましたが，これも16ビットです．ですからスタックにレジスタの内容を入れるときも必ず16ビット単位で行い，オペランドにはBC, DE, HL, IX, IYの各レジスタが指定できます．Aレジスタは8ビットなので，Fレジスタとペアにして AF という16ビットレジスタの形にしてスタックに入れます．

PUSH命令とPOP命令の各ニーモニックは次のようになります．

```
①PUSH  BC
②PUSH  AF
③POP   HL
```

これらの命令が実行されたときのSPの動作は，CALL命令やRET命令が実行されたときとまったく同じです．図7.2にその動作を図解してあるので見てみましょう．

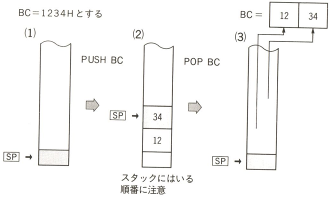



図7.2 スタックへのレジスタの出し入れ

(1)のようにスタックがあるところにBCレジスタをプッシュするとします．Bレジスタの値がSP－1番地，Cレジスタの値がSP－2番地に入れられ，SPは2バイト上がります．この結果スタックは(2)のようになります．また(2)の状態で〝POP　BC〃が実行されると，SP番地の内容がCレジスタに，SP＋1番地の内容がBレジスタにはいって，SPは2バイト下がります．

あるところでBCレジスタをプッシュしておき，(2)のようになったとします．その後BCレジスタを使った演算などを行ってレジスタの内容が壊れてしまっても，また〝POP　BC〃とすれば(3)のようにもとのBCレジスタの値を取ってくることができます．しかしその際，プッシュするときとポップするときのSPの位置が正しく対応していなければなりません．

たとえば(2)の状態から他のレジスタをプッシュしたりサブルーチンをコールしたりすると，スタックは上に伸びてしまいます．そうした場合はそれぞれに対応したところでサブルーチンから戻ったり，プッシュしたレジスタをポップしなければなりません．つまりBCレジスタにポップするときに，(2)のような状態になっていないと，正しいBCレジスタの内容がポップできないのです．

## PUSH／POP命令を利用したサブルーチン

ここでPUSH／POP命令を使って，さきほど出てきたLDBMというサブルーチンを書き換えてみましょう（リスト7.3）．

### リスト 7.3

```
LDBM:   PUSH    HL ……………………HLレジスタの内容をスタックに入れる
        LD      HL,0A020H ……HLレジスタにA020Hをロードする
        LD      (HL),B ……HLレジスタで示されたアドレスのメモリ
                          にBレジスタの内容をロードする
        POP     HL ……………………………………………もとの値をスタックから
                                          HLレジスタに取り出す
        RET ……このサブルーチンを終了してメインルーチンに戻る
```

このようにサブルーチンの中でHLレジスタを使用しますから，それを最初にプッシュしておいて，必要な処理が終わった後でそれをポップします．すると別のルーチンからこのLDBMをコールしたときにHLレジスタの内容は壊れません．BIOSの

〝使用レジスタなし〟というのは，中で使っているレジスタはこのようにPUSH／POP命令によって保存しているという意味なのです．

リスト7.3のHLレジスタは，LDBMをコールする方のルーチンでリスト7.4のようにしても，同じようにHLレジスタを保存したままLDBMをコールすることができます．

**リスト 7.4**

```
PUSH    HL    ………HLレジスタの内容をスタックに入れる
CALL    LDBM ………サブルーチン"LDBM"をコールする
POP     HL    ………もとの値をスタックからHLレジスタに取り出す
```

しかし，この方法はあまりよくありません．まず第1にこのLDBMというサブルーチンは複数のルーチンからコールされる場合，CALL命令のたびにPUSH命令とPOP命令を入れるとプログラム全体が長くなってしまいます．それにこのLDBMというサブルーチンでは，HLレジスタが使われるということを知っていなければなりません．

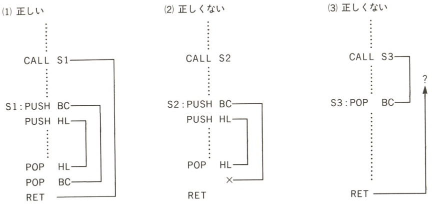
つまり，このようなレジスタの保存はサブルーチンごとに行って，それをコールするメインルーチンには負担をかけないというのが，正しいサブルーチンの作り方です．

## スタックを使うときの注意

このように，スタックというのはサブルーチンを作るときには非常によく使われるものです．しかしこのスタックを，〝サブルーチンのリターンアドレス〟と〝レジスタの保存〟という2つの目的に使用しているため，次のような点に注意する必要があります．

①PUSHとPOPを正しく対応させる
②CALLとRETを正しく対応させる

スタックを使うプログラムを作るときには，まずFILO(First In Last Out の略)のしくみをよく理解し，上にあげた2つの対応を正しくさせることが大切です．図7.3はスタックを使った場合の命令の対応関係をいくつか示したものです．




PUSHとPOPが正しく対応し，RETが〝CALL S1〃と対応する

〝PUSH BC〃に対応するPOP命令がない．こうなると〝PUSH BC〃とRETが対応してしまう

対応のない〝POP BC〃のために，RETの対応がなくなってしまう

**図7.3 PUSH命令とPOP命令を使ったプログラムの例**

まず(1)ですが，これは正しい例です．S1というサブルーチンでは最初に2つのPUSH命令がありますが，最後にPOP命令が2つあって正しく対応しています．S1の中ではSPはいったん2つ分(4バイト)上がりますが，POP命令でこのサブルーチンにはいってきたときの状態に戻すので，RET命令で正しくリターンアドレスに戻れます．

これが(2)の場合にはどうなるでしょうか？ PUSHの部分は同じですが，POP命令が1つ足りません．そのためRET命令を実行するときのスタックの状態は，このS2というサブルーチンにはいってきたときの状態とは違っています．そのためCALL命令でスタックにはいったリターンアドレスではなく，BCレジスタの内容をリターンアドレスとみなしてしまうために，暴走してしまいます．

また(3)ではサブルーチンの最初にリターンアドレスを取り出してしまっているので，後でRET命令が実行されたときに正しいアドレスに戻ることができなくなってしまいます．

## マシンコードの求め方

PUSH 命令，POP 命令はどちらも 1 バイト命令です．ただしインデックスレジスタ IX，IY の場合は，前にそれぞれ DDH，FDH が付きますから 2 バイト命令になります（表 7.1）．

| × | AF | BC | DE | HL | IX | IY |
|---|---|---|---|---|---|---|
| PUSH × | F 5 | C 5 | D 5 | E 5 | D D<br>E 5 | F D<br>E 5 |
| POP × | F 1 | C 1 | D 1 | E 1 | D D<br>E 1 | F D<br>E 1 |

表7.1 PUSH／POP命令の変換表

## Challenge

PUSH／POP 命令を使うと，サブルーチンの中でレジスタの内容を保存することができます．こういうサブルーチンを作っておくと，後でそのサブルーチンを使うときに非常に便利です．

ですから，ここでは〝レジスタの内容を保存する〟サブルーチンの作り方を見ていきます．前にも見ましたが，サブルーチンを作るといろいろなレジスタを使ってしまうため，その内容が壊れてしまいます．ですからサブルーチンを作ったら，その中で使ったレジスタを，サブルーチンの最初でプッシュしておき，RET 命令の前にそれをポップしてください．

### ▶レジスタが壊れるとき

レジスタの内容が壊れるというのはどういう場合に起こるのでしょうか？ 1つはさきほど出てきたように，HL レジスタでアドレスを指定したときです．また BIOS ルーチンをコールするときにも，〝使用レジスタ〟が指定してあれば，そのレジスタの値は壊れます（種々の計算をするときに，レジスタを使用してしまうため）．こういうサ

ブルーチンをコールするときには，レジスタの内容を保存することを考えなければなりません．

さきほどはLDBMというサブルーチンを作ってみたので，ここではレジスタを壊すBIOSルーチンを使います．このルーチンを，レジスタの内容を保存したままコールする方法を考えて見ましょう．

**テーマ** レジスタを保存したまま画面をクリアする

BIOSのCLS(００C３H)というルーチンはエントリーパラメータなしで文字画面をクリアするルーチンですが，AF, BC, DEの各レジスタを使用しています．ですからこれらを保存するプログラムを作ってみます(**リスト**7.5)．

## リスト 7.5

```
D000 018010                LD    BC,01080H    ......BCレジスタに1080H
                                                    をロードする
D003 11C824                LD    DE,024C8H    ......DEレジスタに24C8H
                                                    をロードする
D006 CD12D0                CALL  CLS1         ......サブルーチン"CLS1"
                                                    をコールする
D009 ED4300D8              LD    (0D800H),BC  ......BCレジスタの内容をD800H
                                                    番地のメモリにロードする
D00D ED5302D8              LD    (0D802H),DE  ......DEレジスタの内容をD802H
                                                    のメモリにロードする
D011 C9                    RET                ......モニタに戻る
                           ;
D012 F5           CLS1:    PUSH  AF
D013 C5                    PUSH  BC
D014 D5                    PUSH  DE
D015 CDC300                CALL  CLS          このようにPUSHとPOPとを
D018 D1                    POP   DE           きちんと対応させる
D019 C1                    POP   BC
D01A F1                    POP   AF
D01B C9                    RET
```

まず**リスト**7.5中のCLS1というサブルーチンですが，これは必要なレジスタ(CLSの中で保存されないもの)をプッシュして，CLSをコールした後これらのレジスタをポップしています．つまり外からこのCLS1をコールすれば，レジスタを保存したまま画面をクリアすることができるのです．

このプログラムを〝GD０００″によって実行すると，画面がクリアされ，画面の上にカーソルが表示されているはずです．ここでD８００H番地からのメモリを表示し

てみると図7.4のように１０８０H，２４C８Hという数がはいっているのがわかります．これらの数はCLS1をコールする前に各レジスタに入れられた値ですから，CLS1をコールしてもレジスタの内容が保存されていたということが確かめられたわけです．

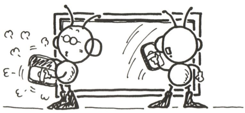
```
>SD000 ⏎
D000 00-01 00-80 00-10 00-11
D004 00-C8 00-24 00-CD 00-12
D008 00-D0 00-ED 00-43 00-00
D00C 00-D8 00-ED 00-53 00-02     プログラムの入力
D010 00-D8 00-C9 00-F5 00-C5
D014 00-D5 00-CD 00-C3 00-00
D018 00-D1 00-C1 00-F1 00-C9
D01C 00-.
>DD000 D01B ⏎
D000 01 80 10 11 C8 24 CD 12   .♠..ネ$ヘ.
D008 D0 ED 43 00 D8 ED 53 02   ミヘC.リヘS.     プログラムの確認
D010 D8 C9 F5 C5 D5 CD C3 00   リノゆナユヘテ.
D018 D1 C1 F1 C9               ムチむノ

>GD000 ⏎ ……………実行する(画面がクリアされる)

          ┌BCレジスタ         ┐……………保存しておいた値がはいっている
>DD800 ⏎  │    ┌DEレジスタ    ┘
D800 80 10 C8 24 00 00 00 00   ♠.ネ$....
D808 00 00 00 00 00 00 00 00   ........
D810 00 00 00 00 00 00 00 00   ........
D818 00 00 00 00 00 00 00 00   ........
D820 00 00 00 00 00 00 00 00   ........
D828 00 00 00 00 00 00 00 00   ........
D830 00 00 00 00 00 00 00 00   ........
D838 00 00 00 00 00 00 00 00   ........

>■
```

図7.4 リスト7.5の実行例



# 7.3

# エクスチェンジ命令

スタックの内容をアクセスする命令には，PUSH／POP 命令がありますが，もう1つスタックに関するものにエクスチェンジ命令というものがあります．これは SP で指されるスタックトップの内容と，レジスタの内容を取り替えるというものです．一種のロード命令のようなものですが，2つのものが同時に書き換わるのが特徴です．

これは④のような形をしています．オペランドには取り替える内容を2つ(ここでは HL レジスタとスタックトップ)を指定します．

④ EX　(SP), HL

これで，HL レジスタの内容と現在のスタックトップの値を取り替えるという命令になります．このとき SP の値は変化しません．

**リスト 7.6** では BC レジスタと HL レジスタの内容の交換を行っています．まず BC レジスタをスタックにプッシュして，そこでエクスチェンジ命令を実行します．この結果 HL レジスタにはスタックトップの1000Hがはいり，HL レジスタにあった2000Hという値はスタックトップにはいります．そこで BC レジスタにポップすれば，2000Hが BC レジスタにはいるというわけです．

**リスト 7.6**

```
D000 010010      LD     BC,1000H
D003 210020      LD     HL,2000H
D006 C5          PUSH   BC         ……BCレジスタの内容(1000H)をスタックに入れる
D007 E3          EX     (SP),HL    ……HLレジスタの内容(2000H)とスタックトップの値を交換
D008 C1          POP    BC         ……BCレジスタにスタックトップの値(2000H)を入れる
D009 ED4300D8    LD     (0D800H),BC
D00D 2202D8      LD     (0D802H),HL
D010 C9          RET
```

実行の結果D８００H番地にはBCレジスタ，D８０２HにはHLレジスタの値がはいりますが，図7.5を見てわかるように最初にロードしたときと両者が入れ替わっています．

```
>SD000 ⏎
D000 00-01 00-00 00-10 00-21 ┐
D004 00-00 00-20 00-C5 00-E3 │
D008 00-C1 00-ED 00-43 00-00 ├ プログラムの入力
D00C 00-D8 00-22 00-02 00-D8 ┘
D010 00-C9 00-.       ─BCレジスタ  ─HLレジスタ
>DD000 D010 ⏎
D000 01 00 10 21 00 20 C5 E3   ...!. ナて  ┐
D008 C1 ED 43 00 D8 22 02 D8   チ\C.リ".リ ├ プログラムの確認
D010 C9                        ノ          ┘

>GD000 ⏎ ……………実行
              ┌BCレジスタ
>DD800 ⏎      │   ┌HLレジスタ
D800 00 20 00 10 00 00 00 00   ........
D808 00 00 00 00 00 00 00 00   ........
D810 00 00 00 00 00 00 00 00   ........
D818 00 00 00 00 00 00 00 00   ........
D820 00 00 00 00 00 00 00 00   ........
D828 00 00 00 00 00 00 00 00   ........
D830 00 00 00 00 00 00 00 00   ........
D838 00 00 00 00 00 00 00 00   ........

>■
```

**図7.5 リスト7.6の実行例**

ふつうスタックを使って16ビットレジスタの内容を交換するときは，

```
PUSH  IX
PUSH  HL
POP   IX
POP   HL
```

というように，PUSH命令とPOP命令を２回ずつ使います．しかしエクスチェンジ命令を使うとスタックに出し入れするのが１回ですむわけです．しかしこのエクスチェンジ命令はHLレジスタとインデックスレジスタでしか行えないため，BCレジスタ

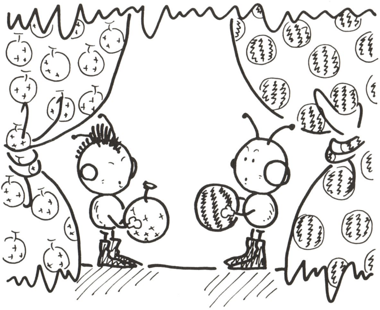
とDEレジスタを入れ替えるということはできません．
ちなみにHLレジスタとDEレジスタの入れ替えには

```
EX    DE,HL
```

という命令があって，スタックをまったく使わずに一度でできる命令があります．
表7.2はエクスチェンジ命令の一部です．

| | | | |
|---|---|---|---|
| EX DE,HL | E B | EX (SP),IX | D D E 3 |
| EX (SP),HL | E 3 | EX (SP),IY | F D E 3 |

**表7.2 エクスチェンジ命令変換表**



# 7.4
# 16進数表示サブルーチン

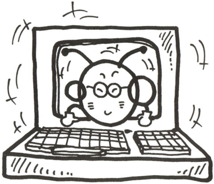
さてここで，今後のために役に立つサブルーチンを作っておきましょう．それはレジスタの内容を16進数で画面に表示するプログラムです．BASICではレジスタの内容を書いたり見たりすることができません．ですからプログラムの実行結果がレジスタにはいっても，最後にはメモリにロードしておかないと，モニタに戻ってきたときにその内容を見ることができませんでした．

これではプログラムがわずらわしくなります．特にこれから先いろいろな演算をレジスタの上で行ったときに，結果はレジスタに残りますが，これをメモリにロードし，モニタに戻ってからいちいちメモリをダンプして見るのはたいへんです．それより，値が出た時点でその値を画面に表示できた方が便利です．

そこでここではAレジスタの内容を16進数で画面に表示するというサブルーチンを作ります．

## PUTA／PUTHLルーチンの作成

**リスト**7.7の中には2つのサブルーチンPUTAとPUTHLがあります．まずPUTAは，Aレジスタの内容(16進数にすると2桁)を計算してキャラクタコードに直し，それをCHPUTで表示します．またPUTHLの方は，このPUTAと同じ方法を使って，HLレジスタの内容(4桁)を16進数で表示します．



## リスト 7.7

```
E000 C5       PUTA:    PUSH   BC     } "PUTHEX"ではA,Cレジスタを使用
E001 F5                PUSH   AF     } しているのでBC,AFレジスタの内容を
                                       スタックに入れる
E002 CD15E0            CALL   PUTHEX ...サブルーチン"PUTHEX"を
                                        コールする
E005 F1                POP    AF     } AF,BCレジスタのもとの内容をス
E006 C1                POP    BC     } タックから取り出す
E007 C9                RET
                       ;
E008 C5       PUTHL:   PUSH   BC
E009 F5                PUSH   AF
E00A 7C                LD     A,H   ……Hレジスタの内容をAレジスタにロードする
E00B CD15E0            CALL   PUTHEX
E00E 7D                LD     A,L   ……Lレジスタの内容をAレジスタにロードする
E00F CD15E0            CALL   PUTHEX
E012 F1                POP    AF
E013 C1                POP    BC
E014 C9                RET
                       ;
E015 4F       PUTHEX:  LD     C,A   ……Aレジスタの内容をCレジスタにロードする
E016 CB3F              SRL    A  }
E018 CB3F              SRL    A  } Aレジスタを右シフトして,上位4ビットの内
E01A CB3F              SRL    A  } 容を下位4ビットの位置にずらす(12章参照).
E01C CB3F              SRL    A  }
E01E CD28E0            CALL   PUTONE
E021 79                LD     A,C
E022 E60F              AND    15   …………下の桁
E024 CD28E0            CALL   PUTONE
E027 C9                RET
                       ;
E028 FE0A     PUTONE:  CP     10         }
E02A 3006              JR     NC,PUTLET  } 0~9であれば画面に表示する
E02C C630              ADD    A,30H      }
E02E CDA200            CALL   CHPUT
E031 C9                RET
                       ;
E032 C637     PUTLET:  ADD    A,37H  } A~Fであれば画面に表示する
E034 CDA200            CALL   CHPUT  }
E037 C9                RET
```

このプログラムの中にはまだ習っていない命令がたくさんありますから，ここでの説明は省きます．とりあえずこのプログラム(E000H番地から始まっていることに注意！)を間違えないように入力してください．

入力ができたら，ここでテストしてみましょう．まずモニタで〝GE000〞を実行すると，2桁の数(16進数)が表示されるはずです．また〝GE008〞とすると，今

度は4桁の数が表示されます．図7.6の実行結果はそれぞれAレジスタとHLレジスタの内容を表示したところです．

```
>DE000 E037 ⏎
E000 C5 F5 CD 15 E0 F1 C1 C9  ナゅヘ.たむチノ
E008 C5 F5 7C CD 15 E0 7D CD  ナゅ¦ヘ.た}ヘ
E010 15 E0 F1 C1 C9 4F CB 3F  .たむチノOヒ?
E018 CB 3F CB 3F CB 3F CD 28  ヒ?ヒ?ヒ?ヘ(     PRINTSのプログラム
E020 E0 79 E6 0F CD 28 E0 C9  たyに.ヘ(たノ
E028 FE 0A 30 06 C6 30 CD A2   .0.ニ0ヘ｢
E030 00 C9 C6 37 CD A2 00 C9  .ノニ7ヘ｢.ノ

>GE000 ⏎ ……………E000H番地からPUTAルーチンを実行する
02 ……………Aレジスタの内容
>GE008 ⏎ ……………E008H番地からPUTHLルーチンを実行する
F7F6 ……………HLレジスタの内容
>■
```

図7.6 リスト7.7実行例

もし本当にレジスタの値を表示したのかどうかを確かめたい場合は，リスト7.8のように，短いプログラムを別のアドレス(たとえばD000Hから)に入力して実行してみてください．

## リスト 7.8

```
LD      A,89H    ……………表示する値
CALL    0E000H
RET
```

```
LD      HL,948FH    ……………表示する値
CALL    0E008H
RET
```

これらのプログラムがうまくいけば大丈夫です．このプログラムは後々使いますから，ここでセーブしておいてください．

# 8章

# 論理演算

# 8.1 論理演算とは

2章でも述べましたが，コンピュータの中のデータというのはすべて2進数の数として表されています．つまりすべてが0か1という世界なのです．このような世界の中で行われる演算を論理演算といいます．この章ではこの論理演算と，これを利用したマスクというものについて説明します．

## 論理演算の種類

論理演算にはたくさんの種類がありますが，代表的なものはNOT(ノット)，AND(アンド)，OR(オア)，XOR(エックスオア)の4つです．ここではこの4つについて説明します．

NOTは否定を行う演算です．0か1しかない世界では，互いに相手は自分の否定ということになり，次のように表現されます．

NOT　1＝0　　（〝1のNOTは0〟と読む）
NOT　0＝1

NOT以外の3つは，足し算や引き算と同じように2つの値から1つの値が決まります．たとえばANDは2つの値が両方とも1のときに1になり，どちらか一方が0なら0になります．ANDのことを論理積といいます．

1　AND　1＝1
1　AND　0＝0

ORは論理和と呼ばれる演算子です．これはどちらか一方が1なら1になり，両方が0のときに0になります．

0　OR　0 = 0

0　OR　1 = 1

XOR（eXclusive（イクスクルーシブ） OR（オア） の略）は排他的論理和といいます．これは一方が 1 で他方が 0 のときに 1 になり，両方が同じ値なら 0 になるというものです．

0　XOR　1 = 1

1　XOR　1 = 0

これら論理演算の働きを表したものが**表 8.1** の〝真理値表〟です．たとえば表の矢印のところを見ると，1 と 0 の OR をとると答は 1 になる，つまり

1　OR　0 = 1

ということがわかります．

| AND | | |
|---|---|---|
| X | Y | Z |
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

| | OR | | |
|---|---|---|---|
| | X | Y | Z |
| | 0 | 0 | 0 |
| | 0 | 1 | 1 |
| ➜ | 1 | 0 | 1 |
| | 1 | 1 | 1 |

| XOR | | |
|---|---|---|
| X | Y | Z |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

| NOT | |
|---|---|
| X | Y |
| 0 | 1 |
| 1 | 0 |

**表8.1　真理値表**

## 論理の世界

ところでいま見たのは，すべてが 1 か 0 かという〝コンピュータの中の世界〟の話でしたが，現実の私達の世界の中で〝論理〟というのはどういう意味があるのでしょうか？

これはまず〝1〟とか〝0〟という 2 つの値を，真と偽にあてはめます．やさしい言葉でいうと，〝ホント〟か〝ウソ〟かということになります．そしていろいろな命題*についてそれが真か偽かを考えます．

＊真か偽の判定ができる問題

たとえば〝リンゴは赤い〟というのは真，〝ミカンは青い〟というのは偽と考えます．ここで否定というのは真偽を反対にすることです．〝ミカンは青い〟のNOTは，偽の反対ですから真になります．

論理積，論理和，排他的論理和というのは，それぞれ〝および〟，〝または〟，〝どちらか一方〟という意味になります．

- ○ **論理積**　　　〝明日は雨および雪が降るでしょう〟
- ○ **論理和**　　　〝明日は雨または雪が降るでしょう〟
- ○ **排他的論理和**　〝明日は雨か雪のどちらか一方が降るでしょう〟

この論理演算というのはBASICのIF文でもしばしば使われます．

```
IF  A=0  AND  B>0  THEN……
```

たとえば上の例では変数Aが0でなおかつ変数Bが正のときに〝真〟になり，THEN以下の文が実行されます．これがORであれば，どちらか一方でも成り立っていれば真になります．

## 論理演算の応用

このように論理演算というのは，現実の世界でものを考えるとよくわかると思います．

それではマシン語を使う場合に，論理演算というのは一体どのように役に立つのでしょうか？ 論理演算はあまり使いようがないように思えますが，以外によく使われるのです．たとえばグラフィックスのプログラムを作る場合には不可欠なものといわれています．また掛け算や割り算に代わるような計算をするのにもよく使われます．

論理演算はマシン語を覚える上では非常に重要です．ですから，よく読んでマスターしてください．前知識としてはここにあげたANDやORの働き方がわかっていれば十分です．

# 8.2

# 論理演算の働き

マシン語での論理演算はAレジスタを使って行われます．論理演算に限らず，演算というのはたいていAレジスタを使います．Aレジスタと他のレジスタやある定数を演算したものが，またAレジスタにはいるというしくみです．論理演算も同様で，Aレジスタ上で8ビットの演算が行われます．

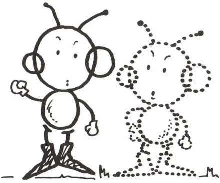
それでは，8ビットの論理演算とは一体どのように行われるのかを見ていきましょう．2進数で8桁の数に論理演算を行うということは，その各ビット(桁)の0，1に対して論理演算を行うということです．例として図8.1を見てください

| | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | ………Aレジスタの値 |
| AND) | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | ………Bレジスタの値 |
| | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | ………ANDの結果 |

**図8.1 8ビットの論理演算**

これは2つの8ビットの数に対してANDという演算を行った結果です．この場合Aレジスタのビット0とBレジスタのビット0で論理積をとり，次にビット1とビット1というようにそれぞれの桁で論理演算を行っているのがわかるでしょう．

NOTやOR，XORも同様で，各ビットごとに演算した結果をとります．



## 論理演算のニーモニック

さて論理演算は，Aレジスタと他のレジスタ，メモリの内容，定数との間で行われ，その結果はまたAレジスタにはいります．つまり，

<Aレジスタ> AND <Dレジスタ> →<Aレジスタ>

ということが行われます．上の<Dレジスタ>のところには8ビットの定数や(HL)に置き換えることもできます．

さて論理演算のニーモニックは，次に示すようにオペランドの部分にAレジスタの演算相手を書きます．これは演算の上ではソースにあたるものです．デスティネーションにあたるものはAレジスタになりますが，書かなくてもわかるのでこのAは書きません．

```
① AND    B
② OR     92H
③ XOR    (HL)
```

①はAレジスタとBレジスタのANDをとって結果をAレジスタに入れ，②はAレジスタと92HのORをとってAレジスタに入れるわけです．

否定のためのNOTという命令はありませんが，CPL(Compliment（コンプリメント）：反転の意味)という命令があってまったく同じことをします．つまり，Aレジスタの各ビットごとに，0であったビットは1に，1であったビットは0になります．この命令は④のようにOPコードのみです．

```
④ CPL
```

## マスク

論理演算の重要な使い道として，マスクというものがあります．これはある数の任意のビットを0にしたり1にしたりするものです．図8.2を見てください．上の数がAレジスタの値，下がマスクパターンと呼ばれるものです．

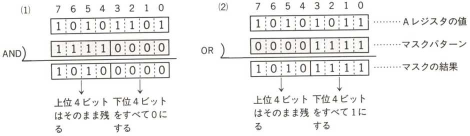



**図8.2 ビットのマスク**

(1)はAレジスタの内容の上位4ビットはそのまま残し，下位4ビットを0にしています．なぜかというと，ある数(0か1)と1のANDをとれば，ある数が1なら1，0なら0になります．つまりもとの数がそのまま残るわけです．またある数と0のANDをとれば，もとの数は必ず0になります．というわけでANDをとればマスクパターンの1のところはそのまま残り，0のところは0にするということができます．

これと同様にORの場合は，ある数と0のORをとればある数が残り，ある数と1のORをとると1になります．(2)は上位4ビットを残し，下位4ビットをすべて1にした例です．

マスクにXORを使うと，マスクパターンの0のビットはそのままにして，1のビットは〝反転する〟ということができます．これは**表8.1**の真理値表を見ながら考えるとわかるでしょう．ある数と0のXORをとればそのままですが，1とのXORをとると0は1に1は0になることがわかります．ですから〝XOR 0FFH〟というのは，Aレジスタのすべてのビットを反転，つまりCPL命令と同じ働きをします．

## マシンコードの求め方

論理演算の命令は，デスティネーションはAレジスタと決まっているので，ソースが何になるかでそれぞれ8種類があります．〝AND 9〟のように，オペランドに定数がくるものは2バイト命令になりますが，それ以外のレジスタや(HL)がオペランドにくるものは1バイト命令になります．

**表8.2**の変換表の見方はロード命令のときと同じです．すなわち〝OR E〟の命令は，〝OR ×〟のEレジスタのところを見て，B3Hを得るわけです．

| × | A | B | C | D | E | H | L | (HL) | (IX+d) | (IY+d) | n |
|---|---|---|---|---|---|---|---|---|---|---|---|
| AND × | A7 | A0 | A1 | A2 | A3 | A4 | A5 | A6 | DD A6 d | FD A6 d | E6 n |
| XOR × | AF | A8 | A9 | AA | AB | AC | AD | AE | DD AE d | FD AE d | EE n |
| OR × | B7 | B0 | B1 | B2 | B3 | B4 | B5 | B6 | DD B6 d | FD B6 d | F6 n |

表8.2 論理演算の変換表

## Challenge

### ▶論理演算の利用

論理演算命令はAND, OR, XORと3種類ありますが，どれも形式は同じです．つまり演算が行われるのはつねにAレジスタで，その相手(オペランドに指定するもの)が他の8ビットレジスタであったり，(HL)や定数であったりするのです．

定数がオペランドになるものは2バイトになりますが，そのほかのものはどれも1バイト命令です．これらは演算の相手にどのレジスタを選んでも結果は同じですから，命令すべてについて例をあげるというようなことはしません．ここでは演算相手はBレジスタだけにしますが，その他のレジスタを使う場合も要領はまったく同じです．

**テーマ** 38Hと6EHのANDをとって結果を表示する

この問題は簡単です．38Hと6EHという2つの値をA，B2つのレジスタに入れ，〝AND　B〞によって論理積をとってやればよいのです．結果はAレジスタに残りますから，そこで前の章で作ったPUTAルーチンをコールすれば*，画面に結果が表示されます．

### リスト 8.1

```
LD      A,38H  ……………Aレジスタに38Hをロードする
LD      B,6EH  ……………Bレジスタに6EHをロードする
AND     B      ……………AレジスタとBレジスタの内容の論理積をとる．結果はAレジスタにはいる
CALL    PUTA   ……………リスト7.7のサブルーチン"PUTA"をコールしてAレジスタの内容を表示する
RET
```

注意　このプログラムは，リスト7.7をメモリにセットしておかなければなりません

＊これ以降のChallengeではリスト7.7のPUTA／PUTHLを使いますから，必ずメモリに入れておいてください．

これをアセンブルしてみましょう．新しい命令としては AND だけです．

## リスト 8.1A

```
D000 3E38          LD      A,38H
D002 066E          LD      B,6EH
D004 A0            AND     B
D005 CD00E0        CALL    PUTA
D008 C9            RET
```

このプログラムを入力して実行した結果が図 8.3 です．

```
>SD000 ⏎
D000 00-3E 00-38 00-06 00-6E  } プログラムの入力
D004 00-A0 00-CD 00-00 00-E0  }
D008 00-C9 00-._              }
>DD000 D008 ⏎
D000 3E 38 06 6E A0 CD 00 E0   >8.n ヘ.た  } プログラムの確認
D008 C9                        ノ          }

>GD000 ⏎……………実行
28 ……………実行結果が表示される
>■
```

図8.3 リスト8.1の実行例

演算の結果，２８Ｈが表示されました．論理演算の結果が正しいかどうかを確かめるには，数を２進数にして考えなければなりません．いまの場合，図 8.4 のような演算を行ったのです．

| | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | ………Aレジスタの値(38H) |
| AND) | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | ………Bレジスタの値(6EH) |
| | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | ………Aレジスタに残る結果(28H) |

図8.4 リスト8.1の演算結果の確認

リスト 8.1 A は，Ｄ００１Ｈ番地とＤ００３Ｈ番地のロード命令のオペランド部分を書き換えると，いろいろな値の AND をとることができます．またＤ００２Ｈ番地のロード命令の OP コードの部分をＢレジスタ以外の８ビットレジスタに変えて，AND

命令のところでそのレジスタとの論理積をとれば，同じことが別のレジスタを使って実行できるのです．

また，このプログラムはORやXORの演算をするときにも使えるので，AND命令(D004H番地)をOR命令やXOR命令に変えて実行してみてください．

### ▶マスクの実行

さて，論理演算を利用して，マスク演算をやってみましょう．

**テーマ**　Cレジスタに適当な値を入れ，その下位4ビットを取り出す

マスクはいろいろ考えられます．テーマであげた以外にも，〝奇数のものは1切り捨てて偶数にする〟とか〝8で割った余りをとる〟とか〝ビットをすべて反転する〟というものもすべてこのマスク演算です．

上のテーマの場合，Cレジスタの値のうち下位4ビットはそのままにして，上位4ビットを0にすればよいのです．ですから図8.5のようにマスク演算子にANDを使い，マスクパターンは0FHになります．

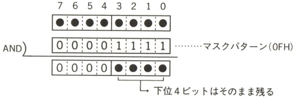



図8.5　0FHをマスクパターンとした場合の結果

これをコーディングしたのがリスト8.2です．

## リスト8.2

```
D000 0E4A       LD      C,4AH      マスクパターン0FHをAレジスタ
D002 3E0F       LD      A,0FH      ……にロードする
D004 A1         AND     C          ……下位4ビットをマスクする
D005 CD00E0     CALL    PUTA       ……マスクの結果"0AH"が表示される
D008 C9         RET
```

リスト8.2を実行してみると，図8.6のように0AHという値が取り出せたことがわかります．D001Hの値をいろいろ変えてみると同じようなことが何度も行えます．いずれも16進数にしたとき上位の桁が0になります．

```
>SD000 ⏎
D000 00-0E 00-4A 00-3E 00-0F
D004 00-A1 00-CD 00-00 00-E0   } プログラムの入力
D008 00-C9 00-.
>DD000 D008 ⏎
D000 0E 4A 3E 0F A1 CD 00 E0   .J>.。ヘ.た  } プログラムの確認
D008 C9                        ノ

>GD000 ⏎ ……………実行
0A ……………下位4ビットだけが取り出せる
>■
```

図8.6 リスト8.2の実行例

今度はマスクパターン(D003H番地)をF0Hに変えてみてください．すると逆に下位の桁が0になり，上位だけが取り出せます．

ところでリスト8.2のように下位4ビットだけを取り出すというのは，Cレジスタの値を16で割った余りを出していることになります．このわけは10進数で考えればすぐわかります．たとえば，34を10で割った余りは1の位だけをとって4になるように，16進数の場合は1の位だけを取り出せば，それは16で割った余りになるわけです．

同じようにAレジスタと7とのANDをとれば8で割った余りがとれますし，1とのANDをとれば2で割った余りがわかるので，奇数か偶数かがわかるのです(図8.7)．

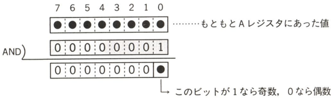



図8.7 マスクを利用した偶数，奇数の判定

次にビットをすべて反転するというプログラムを作ってみましょう．これはＦＦＨをマスクパターンにとって，XOR 命令を行えばよいのです．

**テーマ**　Cレジスタに適当な値を入れて，それをビット反転する

## リスト 8.3

```
D000 0E4A          LD      C,4AH
D002 3EFF          LD      A,0FFH
D004 A9            XOR     C
D005 CD00E0        CALL    PUTA
D008 C9            RET
```

４ＡＨは〝0100 1010〞ですから，これをビット反転すると〝1011 0101〞になりますから，図 8.8 のようにＢ５Ｈが結果として表示されます．

```
>SD000 ⏎
D000 00-0E 00-4A 00-3E 00-FF  ┐
D004 00-A9 00-CD 00-00 00-E0  │ プログラムの入力
D008 00-C9 00-.               ┘
>DD000 D008 ⏎
D000 0E 4A 3E FF A9 CD 00 E0   .J>■ゥヘ.た  ┐ プログラムの確認
D008 C9                        ノ          ┘

>GD000 ⏎ ……………実行
B5 ……………4AHの反転
>■
```

図8.8 リスト8.3の実行例

論理演算を使ってマスク演算をする場合の結果をまとめてみると，マスクパターンの各ビットを０にするか１にするかで，**表 8.3** に示すような作用が起こります．

| マスクパターン ＼ 論理演算 | 0 | 1 |
|---|---|---|
| AND | 0になる | 変わらない |
| OR | 変わらない | 1になる |
| XOR | 変わらない | 反転する |

表8.3 マスクの結果のまとめ

# 9章

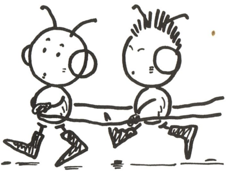
# マシン語の計算

# 9.1
# マシン語の計算方法

BASICに限らず，マシン語のプログラムを作るときにも，計算というのは非常に重要な役割を果たしています．しかしひと口に〝計算〟といっても，ふつう私達が考えるような計算問題ばかりではありません．コンピュータ内部の仕事では，どんな場合にも計算が必要になります．たとえばグラフィック画面に直線を引くというときにも，求める直線に沿う点の座標を１つ１つ計算します．また，単に画面に文字を表示するだけでも，カーソルの位置から文字を表示する場所(V-RAMのアドレス：17章参照)を計算しなければなりません．

Z80のマシン語には，演算をする命令(論理演算は除く)としては，足し算と引き算しかありません．ですからその他の演算，たとえば掛け算や割り算はもちろん，BASICにある三角関数や指数／対数関数のように複雑なものは，これら単純な足し算／引き算命令を組み合せたプログラムを作ってやらなければなりません．

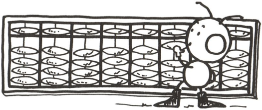
このような理由で，マシン語でプログラムを作るとなると，計算する命令を使う頻度がたいへん多くなります．またこれらを組み合せて使うためには，その機能や働きを十分に頭に入れておく必要があります．



算術演算命令には８ビットと16ビットがあります．この章ではこの２つを順番に見ていきます．

# 9.2
# 8ビットの足し算／引き算

算術演算命令は8ビットのものと16ビットのものがありますが，ここではまず8ビット演算について見ていきます．8ビットの演算は，論理演算と同様Aレジスタを使って行われます．他のレジスタは計算用には使えませんから，計算をたくさん行うプログラムを作るときには，このAレジスタをうまく使わなければなりません．



## 8ビットの足し算

算術演算には足し算と引き算がありますが，まずは足し算から見ていきます．8ビットの演算はAレジスタを使って行われますが，このニーモニックは次のように書きます．

①ADD　A，B
②ADD　A，41H

上のようにオペランドを2つ書き，デスティネーション(1番目)は必ずAレジスタになります．ソースのオペランド(2番目)にはレジスタか定数または(HL)とか(IX+3)というようなメモリ(どれも8ビット)を書き，これをAレジスタに加えるという意味になります．

8ビットの足し算がどう行われるか考えてみましょう．いまAレジスタが68H，Bレジスタが3BHとすると，足し算は図9.1のように行われます．

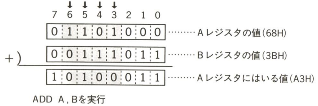



**図9.1 8ビットの足し算**

各桁で足し算をして，1+1になると次の桁に1が繰り上がります．図9.1で数字の上に矢印がついた桁がこの繰り上がりが起こった桁です．

図9.2は繰り上がりの結果，和が8ビットに収まらない場合です．

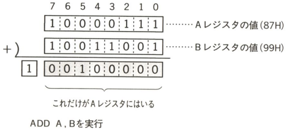



**図9.2 8ビットの足し算で繰り上がりのある場合**

この場合，Aレジスタには20Hだけがはいります．このように最上位で繰り上がりが起こった場合には，Fレジスタに情報がはいります．しかしここではこのことは考える必要はありません．

## 8ビットの引き算

次は8ビットの引き算です．これも足し算と同様に，Aレジスタ上で引き算が行われます．この8ビット引き算命令のニーモニックは次のようになります．

③ SUB　C
④ SUB　0CH

引き算の場合，ニーモニックにはデスティネーションとなる〝A〟は書きません．③はAレジスタからCレジスタの内容を引き，④はAレジスタから0CHという定数を引くということです．このニーモニックの書き方は，論理演算のときと同じですね．

引き算についても，2進数ではどうなるか見てみましょう．**図9.3**は引き算が行われる様子を示しています．

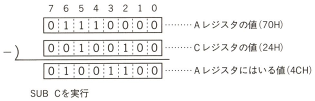



**図9.3 8ビットの引き算**

引き算で小さい数から大きい数を引くと，マイナスの数になります．マイナスの数というのは，8ビットの2の補数表示で表されます．これについてはChallengeのところで実際に実行してみます．

## マシンコードの求め方

演算命令は，ロード命令に比べるとずっと数が少なくなります(**表9.1**)．

```
ADD   A,H……………………… 84
ADD   A,(HL)……………… 86
ADD   A,(IX+08H)……DD 86 08
```

というようになるわけです．オペランドが定数の場合は，

```
SUB   1………………………… D6 01
```

というようになります．

| × | A | B | C | D | E | H | L | (HL) | (IX+d) | (IY+d) | n |
|---|---|---|---|---|---|---|---|---|---|---|---|
| ADD A, × | 87 | 80 | 81 | 82 | 83 | 84 | 85 | 86 | DD 86 d | FD 86 d | C6 n |
| SUB × | 97 | 90 | 91 | 92 | 93 | 94 | 95 | 96 | DD 96 d | FD 96 d | D6 n |

**表9.1 8ビット算術演算命令変換表の一部**

## Challenge

**▶足し算**

ここでは8ビットの足し算／引き算のプログラムを作って，いろいろな計算を実行してみましょう．最初はAレジスタとBレジスタを足すという非常に簡単なものを作ってみます．

**テーマ** 2つの数(47Hと2BH)を足して，結果を表示する

これは最初にA，Bという2つのレジスタに値を入れ，ADD命令によって両者を足します．結果はAレジスタに残りますから，それを画面に表示します(**リスト** 9.1)．

### リスト 9.1

```
D000 3E47        LD    A,47H ……Aレジスタに47Hをロードする
D002 062B        LD    B,2BH ……Bレジスタに2BHをロードする
D004 80          ADD   A,B   ……Aレジスタの内容(47H)とBレジスタの内容(2BH)を足す
D005 CD00E0      CALL  PUTA  ……Aレジスタの内容を表示する
D008 C9          RET
```

**リスト** 9.1を実行したのが，**図** 9.4です．このように47Hと2BHの和の72Hが出力されれば，この足し算はうまくいったことになります．うまくいったら今度は，プログラムのオペランドの部分を書き換えて，いろいろな足し算を試してみてください．

```
>SD000 ⏎
D000 00-3E 00-47 00-06 00-2B  ┐
D004 00-80 00-CD 00-00 00-E0  ├ プログラムの入力
D008 00-C9 00-.               ┘
>DD000 D008 ⏎
D000 3E 47 06 2B 80 CD 00 E0   >G.+♠^.た  ┐ プログラムの確認
D008 C9                        ノ         ┘

>GD000 ⏎……………実行
72 ……………47Hと2BHの和
>■
```

図9.4 リスト9.1の実行例

2つのレジスタの和が8ビット(FFH)を超えた場合は，さきほども述べたように，最上位の9ビット目は無視されて，下位8ビットだけが表示されます．たとえばA8Hと96Hを足すと，

A8H+96H=13EH

となりますが，最上位の1は無視されて，3EHだけがAレジスタにはいります．〝どうして？〟と疑問に思う人もいるかもしれませんが，ここではこうなるのだということを覚えておいてください．詳しいことは次の章で説明します．

▶引き算

次は8ビットの引き算ですが，**リスト9.1**のプログラムをそのまま少し書き換えて引き算のプログラムにしてしまいます．

**テーマ** 2つの数(47Hと2BH)の差を表示する

## リスト 9.2

```
D000 3E47      LD    A,47H
D002 062B      LD    B,2BH
D004 90        SUB   B  ……Aレジスタの内容(47H)からBレジスタの内容(2BH)を引く．結果はAレジスタにはいる
D005 CD00E0    CALL  PUTA
D008 C9        RET
```

これとリスト9.1の足し算プログラムとを比べると，D004H番地だけが違います．ですからいまモニタのSコマンドでメモリに入力してある足し算プログラムを，D004H番地だけ80H(ADD　A，B)から90H(SUB　B)に書き換えればよいのです．

これを実行したのが図9.5です．今度はAレジスタとBレジスタの差が結果として表示されます．これもオペランドを書き換えると，いろいろな引き算ができます．

```
>DD000 D008 ⏎
D000 3E 47 06 2B 80 CD 00 E0   >G.+♠ヘ.た
D008 C9
                    ↑
>SD004 ⏎          "ADD A,B"を"SUB B"に書き換える
D004 [80-90] CD-.
>DD000 D008 ⏎
D000 3E 47 06 2B 90 CD 00 E0   >G.+ ヘ.た
D008 C9                        ノ

>GD000 ⏎ ……………実行
1C ……………実行結果(差)
>■
```

**図9.5 リスト9.2の実行例**

引き算によって，結果がマイナスになってしまう計算をやってみてください．たとえば"1−2"はFFHになります．これは−1(2の補数)という意味になります．

### ▶その他のレジスタ

さて8ビットの足し算／引き算は必ずAレジスタで行われますが，その計算相手はBレジスタに限りません．他の汎用レジスタやHLレジスタ，インデックスレジスタの指すメモリの内容，8ビットの定数が可能です．CからLまでの汎用レジスタを使うには，リスト9.2の2番目のロード命令と演算命令の部分を書き換えるだけでできます．

HLレジスタの指すメモリの内容を使う場合には，その前にHLレジスタにそのメモリのアドレスをロードしなければなりません．また定数の場合には演算命令にオペランドが必要になるので，命令が1バイト長くなります．これは変換表のところでみたとおりです．

# 9.3

# 16ビットの足し算／引き算

さて，今度は16ビットの計算ですが，これは8ビットの場合に比べるとかなり数が少なくなります．演算が行われるレジスタはHL，IX，IYの各レジスタですが，8ビットの場合のように，定数を足したり引いたりすることはできません．

## 命令の種類

まず足し算ですが，足すことのできるレジスタ(ソースにおかれるもの)はBC，DEレジスタ，またはHLレジスタです．定数を足そうと思ったら，いったんこれらのレジスタに足したい定数をロードしてから行います．

⑤ADD　HL，BC

引き算は〝SUB　HL，BC〞という命令がありそうですが，実はありません．16ビットの引き算には，

⑥SBC　HL，BC

という命令があります．これはCYフラグというものがかかわっています．この命令を実行するときには，その前に〝OR　A〞という命令を実行しなければなりません．詳しくは次のChallengeで実際に見ていきます．

## マシンコードの求め方

16ビットの演算は種類が少ないので，マシンコードの表を調べるのは簡単だと思います．特に今回はオペランドに定数がくることはありませんから，マシンコードを求

めるのはそんなにたいへんなことはないでしょう.

| × | BC | DE | HL | SP | IX | IY |
|---|---|---|---|---|---|---|
| ADD HL, × | 09 | 19 | 29 | 39 | | |
| ADD IX, × | DD 09 | DD 19 | | DD 39 | DD 29 | |
| ADD IY, × | FD 09 | FD 19 | | FD 39 | | FD 29 |
| ADC HL, × | ED 4A | ED 5A | ED 6A | ED 7A | | |
| SBC HL, × | ED 42 | ED 52 | ED 62 | ED 72 | | |

表9.2 16ビット算術演算命令変換表の一部

## Challenge

▶ 16 ビットの計算

ここでは HL, DE レジスタを使って, 16 ビットの足し算プログラムを作ってみます. 8 ビットの場合と同様に, 最初に 2 つの 16 ビットレジスタに値を入れ, 演算を行った後 HL レジスタを 16 進数で表示します.

**テーマ** 16ビットどうしの足し算(1892H+689EH)の結果を表示する

## リスト 9.3

```
D000 219218          LD     HL,1892H
D003 119E68          LD     DE,689EH  HLレジスタの値(1892H)と
D006 19              ADD    HL,DE     ………DEレジスタの値(689EH)を
                                         足す. 結果はHLレジスタにはいる
D007 CD08E0          CALL   PUTHL     ………HLレジスタの内容を表示する
D00A C9              RET
```

リスト 9.3 を入力して実行してみたのが図 9.6 です. 1892Hと689EHを足した結果, 8130Hが表示されました. この場合, 足し算の結果は HL レジスタに残りますから, PUTHL でその値を表示しました.

```
>SD000 ⏎
D000 00-21 00-92 00-18 00-11 ⎫
D004 00-9E 00-68 00-19 00-CD ⎬ プログラムの入力
D008 00-08 00-E0 00-C9 00-.  ⎭
>DD000 D00F ⏎
D000 21 92 18 11 9E 68 19 CD  !い..せh.へ ⎫ プログラムの確認
D008 08 E0 C9 00 00 00 00 00  .た/.....  ⎭

>GD000 ⏎ ……………実行
8130 ……………16ビットの和(1892H+689EH)
>■
```

図9.6 リスト9.3の実行例

16 ビット演算の場合，あまり命令のバリエーションがありません．まず定数を直接レジスタに足したりすることはできません．DE レジスタの代わりに BC, HL, SP の各レジスタを足すことができますが，SP はスタックに使われるので，ふつうこのような計算には使いません．また HL レジスタに HL レジスタを加えるというのは，HL レジスタを 2 倍することになります．ですから，ふつうの 16 ビットの足し算は，BC レジスタか DE レジスタに定数なりメモリにあるデータをロードして行います．

また演算が行われるレジスタは HL レジスタの代わりに，インデックスレジスタを使うこともできます．この 2 つのレジスタに関する命令は，HL レジスタの命令の OP コードの前に，それぞれDDH，FDHが付きます．**リスト 9.4** は同じプログラムを IX レジスタを使って行ったものです．

## リスト 9.4

```
D000 DD219218     LD     IX,1892H
D004 119E68       LD     DE,689EH
D007 DD19         ADD    IX,DE
D009 DDE5         PUSH   IX ⎫ IXレジスタの内容をHLレジスタ
D00B E1           POP    HL ⎭ に入れる
D00C CD08E0       CALL   PUTHL
D00F C9           RET
```

IX レジスタの内容はそのままでは表示できませんから，**リスト 9.4** のようにスタックを使って HL レジスタに入れて，PUTHL をコールしています．

次に引き算の命令を使ってみますが，さきほども述べたように〝SUB　HL，DE〞という命令がありません．これと同じ働きをさせるためには

```
OR     A
SBC    HL,DE
```

という２つの命令を使います．最初の〝OR　A〞というのはCYフラグを０にするための命令です．そして下の〝SBC〞という命令は，そのCYフラグを使って演算をするというものです．これについては次の章で詳しく述べますので，いまはこの方法だけを覚えておいてください．

**テーマ**　16ビットどうしの引き算を行う

## リスト 9.5

```
D000 21A398        LD      HL,98A3H
D003 11CF24        LD      DE,24CFH
D006 B7            OR      A ………CYフラグを0にする(10章参照)
D007 ED52          SBC     HL,DE ………16ビットの引き算を行う
D009 CD08E0        CALL    PUTHL
D00C C9            RET
```

```
>SD000
D000 00-21 00-A3 00-98 00-11 |
D004 00-CF 00-24 00-B7 00-ED | プログラムの入力
D008 00-52 00-CD 00-08 00-E0 |
D00C 00-C9 00-.
>DD000 D00C
D000 21 A3 98 11 CF 24 B7 ED  !｣ｸ.ﾏ$ｷ\ | プログラムの確認
D008 52 CD 08 E0 C9           Rﾍ.ﾀﾉ    |

>GD000 ……………実行
73D4 …………16ビットの差(98A3H−24CFH)
>■
```

図9.7 リスト9.5の実行例

# 10章

# Fレジスタの使い方

# 10.1

# Fレジスタの役割

Fレジスタ(フラグレジスタ)については，2章のCPUのときに紹介しましたが，ここでは具体的にどのように使われるのかを説明しましょう.Fレジスタは，アキュムレータや汎用レジスタと同じく裏レジスタを持っている8ビットレジスタですが，他のレジスタとはまったく働きが異なります.

このレジスタにはデータをロードしたり，他のレジスタとの間で演算をすることはできません．演算などのマシン語の命令が実行されたときに，その結果起こった様々なこと(たとえばAレジスタが0になったとか，引き算の結果値が負になった状態など)を保存します.Fレジスタは各ビットごとに違う意味があって,図10.1のような構造になっています.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| S | Z | × | H | × | P/V | N | CY |

S：サイン(符号)フラグ
Z：ゼロフラグ
H：ハーフキャリーフラグ
P/V：パリティ／オーバーフローフラグ
CY：キャリーフラグ
N：加減算フラグ
×：未使用

図10.1 Fレジスタの構造

〝未使用〟というのは，0でも1でも関係ないということです．その他の6つのビットは，演算の結果によって0になったり1になったりします．次の章で説明する条件ジャンプ命令は，この各フラグのビットが0か1かでジャンプするかしないかが決まるのです.

プログラムを作る上で重要なのは，CY，Z，SそれにP／Vの4つのフラグです.

Nフラグと Hフラグは，CPUの内部で使うフラグで，ユーザーが作るプログラムには関係ありません．

Fレジスタの各ビットが1になることを，〝フラグが立つ〞といいます．フラグにはそれぞれ役割があって，マシン語の命令が実行された結果起こった様々なことを意味します．そしてそれらが実際に起こったときに，〝こういうことが起こったぞ！〞ということを示すため，フラグ(旗)が立つのです．

それでは，それぞれのフラグがどういうときに働くのかについて見てみましょう．

## CYフラグ(キャリーフラグ)

CYフラグは足し算や引き算命令が行われたときに，いちばん上の桁で繰り上がりや繰り下がりが起きたかどうかを示します．繰り上がりが起きた場合は1，起きなかった場合は0になります．たとえば，87Hに99Hを8ビット演算で足すと，図10.2に示すようにCYフラグが1になります．また引き算した結果がマイナスになるときも，レジスタには最上位で繰り下がりが起こったものとして，CYフラグは1になります．

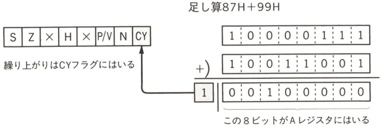



図10.2 繰り上がりが起こる演算の例

ただし，INC／DEC命令は，CYフラグに影響を与えません．たとえば，FFHがはいっているレジスタにINC命令を行うと，そのレジスタは繰り上がりが起こって0になりますが，CYフラグは1にならず，前の状態のまま変わりません．

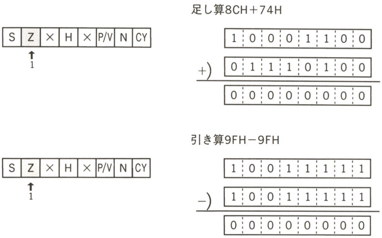
また，CYフラグはAND，OR，XORといった論理演算を行った場合は，必ず0になります．論理演算は各ビットごとに演算が行われるため，桁の外に影響が及ぶということがないからです．

## Z フラグ(ゼロフラグ)

Z フラグは各種演算が行われたときに，A レジスタや HL レジスタに残った結果が 0，つまり，すべてのビットが 0 となったときに 1 になり，それ以外の場合には 0 になります．つまり〝0 のとき 1，0 でないとき 0〞となりますから間違えないようにしてください．

たとえば〝SUB　A〞を実行すると，A レジスタから A レジスタを引くので A レジスタは 0 となり，Z フラグは 1 になります．また足し算の場合も，A レジスタに F F H がはいっているときに，〝ADD　A，1〞を実行すると，A レジスタは 0 になるので Z フラグが 1 になります．論理演算についてもまったく同様で，A レジスタが 0 になったときだけ Z フラグは 1 になります．

また INC／DEC 命令の場合，8 ビットの増減命令(8 ビットレジスタとメモリの内容)のときは影響を受けますが，16 ビットの場合は影響を受けず前の状態がそのまま残ります．




演算の結果が 0 になった場合 Z フラグは 1 になる

図10.3　Z フラグが 1 になる演算の例

## Sフラグ(サインフラグ)

Sフラグのサインというのは符号(Sign)という意味です．これは演算の結果Aレジスタや HL レジスタに残った値が，正(+)であるか負(−)であるかを示します．ですからこのフラグは，Aレジスタなら8ビット，HLレジスタなら16ビットの2の補数表示で考えているときにしか意味がありません．

図10.4を見てください．これはAレジスタを使った演算を行ったときのSフラグの受ける影響を表したものですが，このようにAレジスタのビット7がそのままSフラグにはいります．なぜならAレジスタを2の補数表示で考えると，ビット7が0なら正，1なら負になるからです．

ですから計算の結果Sレジスタが0なら正，1なら負の答が出たということがわかります．同じ計算を行っても，(2)のように符号なしの8ビットの正の数と考えているときは，Sフラグの結果には(もちろん同じ影響を受けますが)意味がありません．

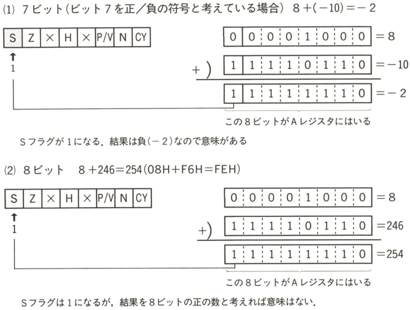
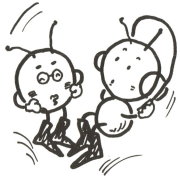



図10.4 Sフラグの意味

## P／V フラグ(パリティ／オーバーフローフラグ)



P／V フラグの働きには，パリティフラグとしてのものとオーバーフローフラグとしてのものの2つがあります．まずパリティフラグとして働くのは論理演算を行った場合です．これは AND や OR をとった結果，A レジスタの8ビットのうち1となっているビットの数が偶数個のとき1，奇数個のとき0になります．

またオーバーフローフラグとして働くのは算術演算命令のときですが，S フラグと同様に A レジスタの値を2の補数表示として扱っているときにしか意味がありません．P／V フラグは演算結果が－128から＋127までの範囲から出てしまったときに1，そうでないとき0となります．

```
3FH＋67H (63＋103)
     3 F H
 +)  6 7 H
 ─────────
     A 6 H(＝－90)
```

……この場合オーバーフローフラグとして働き，足し算の結果(166)は＋127を超えるためにP／Vフラグは1になる．2の補数の計算では，P／Vフラグが1になった場合正しい答えが得られていないことを意味する．

図10.5 P／Vフラグ

以上，ユーザーがプログラムの中でかかわることのできるフラグは4つです．これらがFレジスタの中で何ビット目にあるかということは，あまり意味がありません．むしろそれぞれのフラグがどういう働きをするかということの方が大切です．

プログラムの中でよく使われるのは Z フラグと CY フラグの2つです．S フラグは補数表示(8ビットを符号付き7ビットとみる)で扱っているときでないと使いませんし，P／V フラグの場合も同じです．

# 10.2
# フラグに関する命令

フラグは，その多くが条件判断で用いられますが，これについては次の節の比較命令と，次の章の条件ジャンプ命令のところで説明します．もう1つCYフラグを応用したものとして，ADC（アドキャリー）／SBC（サブキャリー）という命令がありますので，ここではそれについて説明しましょう．

CYフラグは演算を行ったときに，いちばん上の桁で繰り上がりや繰り下がりが起こったときに立つフラグですが，これを利用すると複数のバイトにまたがる数値の演算を行うことができます．

たとえば**図10.6**は2バイトと2バイトの足し算をする例ですが，ふつう私達が行うようにまず下位の桁(バイト)を足します．するとここで繰り上がりが起こりますから，この繰り上がった1は次に上の桁(バイト)どうしを足すときに，そこに足さなければなりません．

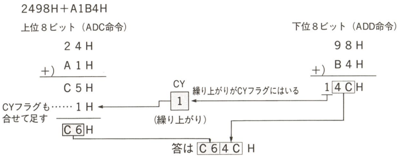



**図10.6 2バイトの足し算**

図10.6の足し算を，8ビットのAレジスタを使って行うことを考えてみましょう．まず下位8ビットどうしを足すと答は4CHとなって，繰り上がった1はCYフラグ

にはいります．次に上位8ビットどうしを足しますが，このとき前に行った下位の足し算の繰り上がり分がはいってこなければなりません．ですから，単にオペランドにあるレジスタの値や定数を足すだけでなく，〝CY フラグを合せて足す〞という命令が必要になります．

このようにCY フラグを合せて足すという命令が次のADC命令です．

①ADC　A，C
②SBC　A，34H
③ADC　HL，DE
④SBC　HL，BC

①と③がCY フラグを合せて足すという演算です．ニーモニックの書式はADD命令のときと同じです．②と④はCY フラグを含めた引き算命令です．ただし②の書式はSUB命令とは違って，デスティネーションとなるAレジスタも書きますから注意してください．

ADC命令を使って図10.6の計算を行うプログラムを作ってみると，リスト10.1のようになります．

## リスト 10.1

```
LD      A,98H    ……………Aレジスタに98Hをロードする
ADD     A,0B4H   ……………ここでCYフラグが1になる.
LD      C,A    } 下位の8ビットはCレジスタに入れる
LD      A,24H  } ロード命令ではフラグは変化しない
ADC     A,0A1H   ……………ここでCYフラグもいっしょに足される
LD      B,A      ……………Aレジスタの内容はC6H．上位8ビットをBレジスタに入れる
```

下位の足し算命令と上位の足し算命令の間にロード命令が2つはいっていますが，これらはフラグに影響を与えませんから，ADC命令のところでうまくCY フラグを使うことができるのです．このようにして，演算と演算の間でCY フラグの状態が変わらないようにすれば，何バイトの演算でもできます．

また，②,④のCY フラグを含めた引き算は，引き算を行ってCY フラグも合せて引くという命令です．図10.7が，このSBC命令を使って2バイトの数値から2バイトの数値を引いた演算です．

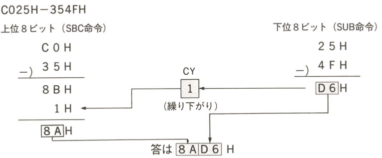


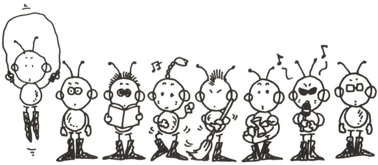

図10.7　2バイトの引き算

このプログラムは**リスト 10.1** とほとんど同じになりますから，ここでは省略します．もちろんこの場合も下位８ビットの引き算を先に行います．

なお③と④は，16 ビット演算で，働きは８ビットの場合と同じです．16 ビットの引き算命令は SUB 命令がないので，SBC 命令を使わなければなりません．しかし，そのためには CY フラグを０にしておかなければなりませんから，それには引き算の前に〝OR　A″を実行しておきます．

```
OR      A ……………… A レジスタの内容を変えずに CY フラグを 0 にする
SBC     HL，BC … CY フラグは 0 なので，その影響は受けない
```

こうすれば，CY フラグは０となり，SBC 命令を使っても SUB 命令と同じ結果が得られるのです．また〝OR　A″を実行しても，A レジスタの内容は変わりません．



## マシンコードの求め方

この節で出てきた命令は，ADC 命令と SBC 命令の 2 つです．**表 10.1** に 8 ビット演算の命令の変換表を示しておきます．

| × | A | B | C | D | E | H | L | (HL) | (IX+d) | (IY+d) | n |
|---|---|---|---|---|---|---|---|---|---|---|---|
| ADC A, × | 8F | 88 | 89 | 8A | 8B | 8C | 8D | 8E | DD 8E d | FD 8E d | CE n |
| SBC A, × | 9F | 98 | 99 | 9A | 9B | 9C | 9D | 9E | DD 9E d | FD 9E d | DE n |

**表10.1 ADC, SBC 8 ビット変換表**

16 ビットの ADC 命令，SBC 命令は 16 ビット変換表にまとめて載っています．こちらは数が少なく，それぞれ 4 つずつ，合せて 8 つしかありません(**表 10.2**)．

| × | BC | DE | HL | SP |
|---|---|---|---|---|
| ADC HL, × | ED 4A | ED 5A | ED 6A | ED 7A |
| SBC HL, × | ED 42 | ED 52 | ED 62 | ED 72 |

**表10.2 ADC, SBC 16ビット変換表**

## フラグへの影響

さて，さきほどの 2 バイトどうしの足し算のプログラムを作ったときに，〝ロード命令ではフラグに影響を与えない〟といいましたが，このようにマシン語の各命令が，実行の結果どのフラグにどんな影響を与えるかということは，フラグを扱うプログラムを作る上で非常に重要です．

巻末付録の変換表の中に，**表 10.3** のような各命令がフラグに与える影響を書いたものがありますので，この表の見方を説明しておきます．

| インストラクション | SZ H P/V N CY | コメント |
|---|---|---|
| ADD A,s ; ADC A,s | ↕↕×↕×V0↕ | 8ビット足し算命令 |
| SUB s ; SBC A,s ; CP s ; NEG | ↕↕×↕×V1↕ | 8ビット引き算・比較・NEG命令 |
| AND s | ↕↕×1×P00 | 論理演算命令 |
| OR s ; XOR s | ↕↕×0×P00 | |
| INC | ×↕×V0・ | 8ビット増減命令 |
| ; LDD | ・・×0× | |
| LDIR ; LDDR | ・・×0×00・ | |
| CPI ; CPIR<br>CPD ; CPDR | ↕↕×××↕1・ | ブロックサーチ命令<br>A=(HL)ならばZ=1, その他はZ=0<br>BC−1≠0ならばP/V=1, その他はP/V=0 |
| LD A,I ; LD A,R | ↕↕×0×IFF0・ | IFFの内容をP/Vにコピー |
| BIT b,s | ×↕×1××0・ | Sのビットbの内容をZにコピー |

表10.3 変換表の一部

ここには，S，Z，H，P/V，N，CYの各フラグが，それぞれの命令によってどんな影響を受けるかが書かれています．表には，〝↕，・，1，0〟という4つの記号のどれかが書かれていますが，それぞれの意味は下に示すとおりです．

- ↕ ： 演算の結果によって0または1になる
- ・ ： フラグは影響を受けない
- 1 ： 1になる
- 0 ： 0になる

また，P/Vフラグのところでpとかvの文字が出ているのは，それぞれPならパリティフラグとして，Vならオーバーフローフラグとして影響を受けるという意味です．

## Challenge

### ▶複数バイトの演算

この節では，CYフラグを使った演算を行うADC命令とSBC命令について見てきました．これらの命令を使うと，何バイトにもまたがる数値の足し算や引き算をすることができます．

**テーマ** BCレジスタとDEレジスタの和を計算する

16 ビットの値は上位 8 ビットが B レジスタと D レジスタに，下位 8 ビットが C レジスタと E レジスタにそれぞれはいっています．これを足し合せるには上位は上位，下位は下位で足せばよいのですが，下位どうしを足したときに繰り上がりが起きた場合に，この繰り上がり分は上位 8 ビットに加えられなければなりません．

リスト 10.2 では，まず下位 1 バイトの C レジスタと E レジスタを足します．この例ではここで CY フラグが 1 になり，これが D 0 0 C H番地の足し算"ADC A，D"で上位バイトに加えられます．このとき注意したいのは，ADD 命令から次の ADC 命令の間にある 2 つのロード命令は，CY フラグに変化を与えないということです．

## リスト 10.2

```
D000 0164A8       LD    BC,0A864H
D003 11BD24       LD    DE,24BDH
D006 79           LD    A,C
D007 83           ADD   A,E ..............下位8ビットの足し算(ここで
                                          CYフラグが1になる)
D008 3200D8       LD    (0D800H),A  } ロード命令ではCY
D00B 78           LD    A,B         } フラグは変化しない
D00C 8A           ADC   A,D ..............上位8ビットの足し算
D00D 3201D8       LD    (0D801H),A
D010 C9           RET
```

```
>SD000 ⏎
D000 FF-01 FF-64 FF-A8 FF-11  }
D004 FF-BD FF-24 FF-79 FF-83  }
D008 FF-32 FF-00 FF-D8 FF-78  } プログラムの入力
D00C FF-8A FF-32 FF-01 FF-D8  }
D010 FF-C9 FF-.               }
>DD000 D010 ⏎
D000 01 64 A8 11 BD 24 79 83  .dィ.ス$y◆  }
D008 32 00 D8 78 8A 32 01 D8  2.リxぇ2.リ  } プログラムの確認
D010 C9                       ノ          }

>GD000 ⏎ ..............実行

>DD800⏎ ...........結果を確認してみる
                 ──この2バイトで，答はCD21Hであることがわかる
D800 21 CD 00 00 00 00 00 00  !へ......
D808 00 00 00 00 00 00 00 00  ........
D810 00 00 00 00 00 00 00 00  ........
D818 00 00 00 00 00 00 00 00  ........
```

図10.8 リスト10.2実行例

これを実行してみたのが図10.8です．ご覧のように答はみごとCD21Hになりました．上位8ビットがCCHではなく，CDHになっています．このプログラムは最初のロード命令のオペランドを書き換えれば，いろいろな足し算ができます．また16ビットの引き算をする場合には**リスト10.3**のように足し算命令を引き算命令に変更します．

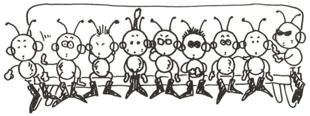
## リスト 10.3

```
D000 0164A8      LD    BC,0A864H
D003 11BD24      LD    DE,24BDH
D006 79          LD    A,C         下位8ビットの引き算（CYフ
D007 93          SUB   E  ·········ラグが1になる）
D008 3200D8      LD    (0D800H),A  ｝ロード命令ではCY
D00B 78          LD    A,B         ｝フラグは変化しない
D00C 9A          SBC   A,D ··········上位8ビットの引き算
D00D 3201D8      LD    (0D801H),A
D010 C9          RET
```

**リスト10.2**や**リスト10.3**ではADC命令やSBC命令の働きを見るために，16ビット演算にAレジスタを2回使っていますが，実際には16ビット演算は**リスト10.4**にあるように，HLレジスタを使えば演算命令は1つですみます．

## リスト 10.4

```
LD     BC,0A864H
LD     DE,24BDH
LD     H,B   ｝演算するためにHLレジスタにロードする
LD     L,C   ｝
ADD    HL,DE ·························引き算の場合は"OR A","SBC HL,DE"とする
CALL   PUTHL
RET
```



# 10.3

# 比較命令

前の節で出てきたADC／SBC命令は，CYフラグに関するものでしたが，フラグにはこのほかにもZ，S，P／Vなどがあります．フラグの本来の働きというのは，次の章に出てくる〝条件ジャンプ″の際に，IF〜THEN文のような条件分岐を実行するためのものです．

この章ではまだ条件ジャンプを説明しないので，条件分岐をするプログラムを作ることはしませんが，ここではその準備として，これらフラグに影響を与える比較命令というものを紹介しておきましょう．

フラグが最もよく使われるのは，2つの数を比較する場合です．たとえば，Aレジスタの内容とBレジスタの内容を比べるときは，〝SUB　B″とします．すると，もし2つのレジスタの内容が同じならば，引き算の結果Aレジスタは0となってZフラグが立ちますし，もしBレジスタの方が大きければCYフラグが立ちます．

しかし，SUB命令を実行すると，どうしてもAレジスタの内容が変わってしまいます．これを変えずに，単に比較だけをするためには，次に示すCP(コンペア)命令というものを使います．

⑤ＣＰ　　Ｂ
⑥ＣＰ　　３

⑤は〝＜Aレジスタ＞−＜Bレジスタ＞″，⑥は〝＜Aレジスタ＞−3″という演算を行いますが，結果をAレジスタに入れません．ただZフラグやCYフラグなど，フラグに与える影響はSUB命令とまったく同じものです．

このCP命令は，通常条件ジャンプ命令と組み合せて使います．ここではまだそのジャンプ命令を習っていないので，実際の使い方を示すことができません．CP命令に関するChallengeは，次の章で条件ジャンプ命令を習ってからにしましょう．

# 11章
# プログラムの制御

# 11.1
# ジャンプ命令とプログラム

この章で説明するジャンプ命令というのは，プログラムを作る上で最も重要な制御構造*を実現するものです．

ジャンプ命令はBASICのGOTO文にあたる命令で，プログラムの実行順序を変える働きをするものです．この命令には，CPUがその命令に出会ったときに必ずジャンプするもの(無条件ジャンプ)と，ある条件が成り立ったときにジャンプをするもの(条件ジャンプ)の2通りがあります．

前の章でも触れたように，条件ジャンプはその命令が実行されたときのフラグの状態によってジャンプするかどうかが決まるため，演算命令や比較命令といった，フラグに影響を与える命令と組み合せて使われます．この〝演算＋条件ジャンプ〟という組合せによって，BASICの〝IF～THEN〟のような条件分岐ができるようになります．

しかしBASICの〝IF～THEN〟や〝ON　X　GOTO〟のようなものを作ろうと思ったら，ジャンプ命令をいくつも組み合せなければなりません．〝ジャンプ命令をうまく使えるか〟ということは，プログラム作りがうまいかどうかの尺度になります．ですから人のプログラムを見るときは，条件ジャンプをどう使っているかということを注意して見ましょう．

この章ではジャンプ命令を，**絶対アドレスジャンプ**と**相対アドレスジャンプ**という2つの種類に分けて見ていきます．どちらもジャンプをするのは同じですが，実行するときの動作が異なるので，アセンブルのときのマシンコードの出し方が異なります．

それから最後に，DJNZ命令というものを説明します．これはジャンプ命令の一種ですが，ループを実現するための命令です．

---

＊プログラムの流れ(実行順序)をコントロールするもの

# 11.2

# 絶対アドレスを指定するジャンプ

絶対アドレスジャンプ，相対アドレスジャンプの両方に，無条件ジャンプと条件ジャンプがあります．ここでは絶対アドレスジャンプを説明しますが，まずは無条件ジャンプから見ていきましょう．

ジャンプ命令というのはプログラムの実行順序を変える命令です．BASICのGOTO文ではジャンプ先の行番号を指定したように，マシン語のジャンプ命令ではジャンプ先のメモリのアドレスを指定します．これはCALL命令と同じです．

絶対アドレスジャンプのニーモニックは①のように書きます．

```
① JP  9823H
```

この命令を実行すると，９８２３H番地にジャンプして，そこにある命令を実行します．CPUはこのジャンプ命令に出会うと，オペランド(ジャンプ先のアドレス)をプログラムを読み出すためのレジスタPCに入れるのです．

## ジャンプ命令のしくみ

CALL命令のときにも述べましたが，このPCというレジスタはプログラムのあるメモリのアドレスを指しています．CPUはPCの指すアドレスから命令を読み出して，PCを次の命令のところまで進めた後，読み出した命令を実行します．

たとえば①の命令が９０００H番地にあったとします．この命令が読み出されると，PCは次の命令のあるアドレス(①のジャンプ命令は３バイト命令なので９００３H)に進みます．しかしここで読み出したジャンプ命令を実行すると，PCが９８２３Hに変わります．次に実行する命令はこのPCの指すアドレスから読み出しますから，これで９８２３H番地にジャンプすることができたことになるのです(図11.1)．

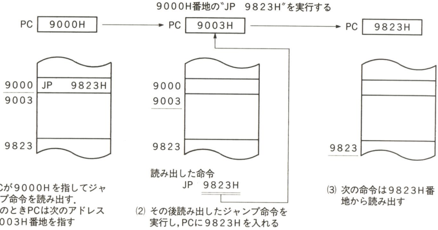



図11.1　ジャンプ命令が実行される様子

## ラベルによるジャンプ先アドレスの指定

ジャンプ命令を使ったプログラムを書く場合，どこにジャンプするにしてもそのジャンプ先のアドレスを指定しなければなりません．しかしプログラムは最初ニーモニックで書きますから，その時点では各命令が何番地にはいるかということがわかりません．これはアセンブルしたときに，命令を1つ1つアドレスにあてはめていって初めてわかるのです．

そのためニーモニックでプログラムを書くときは，図11.2にあるように，ジャンプ先の命令のところにラベルを付けておきます．そしてジャンプ命令のオペランドに同じラベルを書くのです．

```
                  :
         JP       KEYTS2 ……………ジャンプ命令のオペランドにはラベルで指定
         LD       A,41H
         CALL     CHPUT
KEYTS2:  RET
  :……ジャンプ先のラベル  :
```

図11.2　ジャンプ命令でのラベルの使用

## 条件ジャンプ

次に条件ジャンプについて見てみます．条件ジャンプはフラグの状態(0か1か)によってジャンプするかどうかが変わるという命令で，ニーモニックは次のような形をしています．

```
② JP    C,LABEL2
③ JP    NZ,LABEL2
```

上のように，JPとラベルの間に〝C〞とか〝NZ〞という文字と，カンマがはいっています．この文字が〝どういうときにジャンプするか〞という条件を示しているのです．これはあるフラグに注目して，それが1か0かを指定するもので，全部で次の8つがあります．

○C　　CYフラグが1のとき
○NC　　CYフラグが0のとき
○Z　　Zフラグが1のとき
○NZ　　Zフラグが0のとき
○PE　　P／Vフラグが1のとき(パリティがイーブン)
○PO　　P／Vフラグが0のとき(パリティがオッド)
○M　　Sフラグが1のとき(マイナス)
○P　　Sフラグが0のとき(プラス)

これらの条件をもとに②，③の例を見てみると，②は〝CYフラグが1ならLABEL2のアドレスにジャンプせよ〞，③は〝Zフラグが0ならLABEL2のアドレスにジャンプせよ〞という意味になります．どちらの場合も，指定されたフラグの状態を調べ，それがニーモニックに指定された条件に合えば指定されたアドレスにジャンプします．もしフラグの状態が逆のときは，ジャンプせずにその次の命令を実行します．

前にも述べましたが，条件ジャンプは比較や演算命令と組み合せて使います．たとえば**リスト11.1**は，〝Aレジスタの値が40Hより小さければLABEL1にジャンプしろ〞というプログラムになります．

## リスト 11.1

```
CP    40H       } IF A<40H THEN JP LABEL という意味
JP    C,LABEL   } Aレジスタが40Hより小さいと，CP命令でCYフラグが立つ
```

リスト 11.1のCP命令を実行したとき，Aレジスタにどんな値がはいっていればCYフラグが1になるかを考えてみてください．こういう場合，Aレジスタにちょうど40Hがはいっていた場合，どうなるかも考えなければなりません(CYフラグは立たないのでジャンプしませんよ！)．もしも〝40H以下ならジャンプせよ〞としたければ，比較する値を41Hにしなければなりません．

## マシンコードの求め方

絶対アドレスジャンプ命令は3バイト命令になり，そのオペランドは2バイトのアドレスです．この命令の形はCALL命令のときと同じです．つまりオペランドとなるアドレスは，例によって上位と下位を逆にします．たとえば，

```
JP  9822H
```

というのは，JPのOPコードがC3Hですから〝C3，22，98〞になります．その他の条件ジャンプのOPコードなどは，表11.1を参照してください．

| × | 無条件 | キャリー | ノン キャリー | ゼロ | ノン ゼロ | パリティ 偶数 | パリティ 奇数 | 負 | 正 |
|---|---|---|---|---|---|---|---|---|---|
| | | C | NC | Z | NZ | PE | PO | M | P |
| JP ×,nn | C3<br>n<br>n | DA<br>n<br>n | D2<br>n<br>n | CA<br>n<br>n | C2<br>n<br>n | EA<br>n<br>n | E2<br>n<br>n | FA<br>n<br>n | F2<br>n<br>n |

**表11.1 ジャンプ命令変換表**

絶対アドレスジャンプ命令の変換表には，オペランドが〝nn〞と書いてありますが，これはジャンプ先のアドレスを2バイトで指定するという意味です．もちろん下位，上位の順にすることを忘れてはいけません．

## 条件コールと条件リターン

フラグの状態によってジャンプするかしないかが決まる条件ジャンプ命令があるように，前に出てきたCALL命令とRET命令にも条件命令があります．ニーモニックの書式はほとんど同じです．

```
④CALL    Z,LABEL4
⑤RET     C
```

どちらもそのフラグの条件が成り立っているときはコールまたはリターンを実行し，そうでないときはその次の命令を実行します．

これら条件コール，条件リターンのマシンコードは，前に説明したCALL命令の変換表を参照してください．

## Challenge

**▶条件判定を行う**

ここでは絶対アドレスジャンプ命令を使ったプログラムを作ってみます．プログラムそのものはもちろん，ジャンプ命令のオペランドの決め方も大切です．アセンブルの過程にも注目してください．

また，ここでは前の章で説明したCP命令を使ってみます．これはプログラムとしてはここで初めて出てくるものです．その働きについてはちゃんと覚えていますか？ 比較の結果，フラグがどのように影響を受けるかということがポイントになります．

**テーマ** 押されたキーによって2通りの文字を画面に表示する

**リスト11.2**は押されたキーを調べて，その種類によって画面に表示する文字を決めるというプログラムです．仕様は次のように決めておきます．

- ○ 0～9の数字キーが押されたら，〝N〟(NUMBERの頭文字)を表示する
- ○ それ以外のキーが押されたら，〝A〟(ALPHABETの頭文字)を表示する

押されたキーを調べるには，前に使ったCHGETというBIOSルーチンを使います．これによってAレジスタにキャラクタコードがはいるので，CP命令によってそのキャラクタコードを調べます．

## リスト 11.2

```
KEYTST: CALL    CHGET
        CP      30H          ……………"0"のキャラクタコード
        JP      C,KEYTS1     ……………キャラクタコードが30H未満のもの
        CP      3AH          ……………":"のキャラクタコード
        JP      NC,KEYTS1    ……………キャラクタコードが3AH以上のもの
        LD      A,4EH        } "N"を表示する
        CALL    CHPUT        }
        JP      KEYTS2       ……………合流する
KEYTS1: LD      A,41H        } "A"を表示する
        CALL    CHPUT        }
KEYTS2: RET
```

**リスト** 11.2では，CHGETをコールした後でプログラムが2つに分かれて，また最後にKEYTS2のラベルのところで合流しています．まずAレジスタにはいったキャラクタコードが，３０Hから３AHの間であれば入力された文字が数字であることがわかります．そこでこの範囲にはいっていない場合はKEYTS1にジャンプして〝A〟を表示します．

BASICならば，

```
IF C<&H30 OR C>=&H3A THEN ……
```

と条件式の途中に論理演算子ORを使って１つのIF文で分岐をするということができますが，マシン語では少しでも複雑な分岐をしようとしたら，**リスト** 11.2のように〝比較＋条件ジャンプ〟という組合せをいくつも書かなければなりません．

もし〝８〟というキーを押したときは何という文字が表示されるかということを調べたいなら，Aレジスタに３８Hという値がはいったときはどこにジャンプするかというプログラムの流れを追ってみるとよいでしょう．

次にこのプログラムをアセンブルしてみます．

このプログラムではKEYTS1とKEYTS2という2つのラベルがあります．どちらもラベルがJP命令自身よりも下にありますから，最初にこれらをアセンブルするときはオペランドを2バイト空けておかなければなりません．後でそれぞれのラベルの

アドレスが決まったらオペランドを入れます.

## リスト 11.2A

```
D000 CD9F00     KEYTST:  CALL    CHGET
D003 FE30                CP      30H
D005 DA15D0              JP      C,KEYTS1
D008 FE3A                CP      3AH
D00A D215D0              JP      NC,KEYTS1
D00D 3E4E                LD      A,4EH
D00F CDA200              CALL    CHPUT
D012 C31AD0              JP      KEYTS2
D015 3E41       KEYTS1:  LD      A,41H
D017 CDA200              CALL    CHPUT
D01A C9         KEYTS2:  RET
```

最初は2バイト分空けておく

ジャンプ先アドレスが決定した後，空けておいたところに書き込む

このプログラムを実行すると，最初は何も表示されません．これはキー入力を待っている状態です．ここで何かキーを押すと図 11.3 のように，押したキーが数字であれば "N"，それ以外なら "A" という文字が表示されて，プログラムは終わります(モニタに戻ります)．何度でも実行できますから，いろいろなキーを押してみてください．

```
>SD000
D000 FF-CD FF-9F FF-00 FF-FE
D004 FF-30 FF-DA FF-15 FF-D0
D008 FF-FE FF-3A FF-D2 FF-15
D00C FF-D0 FF-3E FF-4E FF-CD      プログラムの入力
D010 FF-A2 FF-00 FF-C3 FF-1A
D014 FF-D0 FF-3E FF-41 FF-CD
D018 FF-A2 FF-00 FF-C9 FF-.
>DD000 D01F
D000 CD 9F 00 FE 30 DA 15 D0   ヘ〒.  0レ.ミ
D008 FE 3A D2 15 D0 3E 4E CD    :メ.ミ>Nヘ    プログラムの確認
D010 A2 00 C3 1A D0 3E 41 CD   「.テ.ミ>Aヘ
D018 A2 00 C9 FF FF FF FF FF   「./■■■■■

>GD000
N ・・・・・・・・・・5を押すと"N"が表示される
>GD000
A ・・・・・・・・・・Sを押すと"A"が表示される
>■
```

図11.3 リスト11.2A実行例

# 11.3

# 相対アドレスを指定するジャンプ

さてジャンプ命令には，もう１つ〝相対アドレスジャンプ〟というものがあります．絶対アドレスジャンプでは，オペランドにジャンプ先のアドレス（２バイト）を書きましたが，相対アドレスジャンプ命令というのは，オペランドが１バイトしかなく，そこに相対アドレスというものを書きます．

相対アドレスというのは，ジャンプ先に行くためにはPCがどれだけ動けばよいかということを意味します．後で詳しく説明しますが，相対アドレスを使うと，ハンドアセンブルするときに計算が必要になります．

## ニーモニックの書き方

相対アドレスジャンプにも，無条件ジャンプと条件ジャンプがあります．OPコードは〝JR〟(Jump Relative の略)になりますが，ニーモニック自体の書式は絶対アドレスジャンプと変わりません．

```
④JR   LABEL
⑤JR   Z,LABEL
```

④は無条件ジャンプ，⑤はZフラグによる条件ジャンプです．OPコードは〝JR〟になりますが，オペランドにはラベルを書きます．絶対アドレスのときはそのラベルのアドレスそのものをオペランドにしました．しかし相対アドレスジャンプ命令をアセンブルするときは，〝JR命令のあるアドレスからラベルのアドレスまで何バイト離れているか〟ということを書きます．これが相対アドレスというものです．

## 相対アドレスの求め方

相対アドレスジャンプのマシンコードは**図 11.4** のようにして求めます．まず(1)がソースプログラムです．最初に JR 命令がありますが，オペランドになっているラベル LS4 のアドレスがまだわかりません．そこで例によって，オペランドの部分を 1 バイトだけ空けておいて次に進みます．

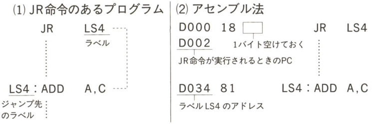


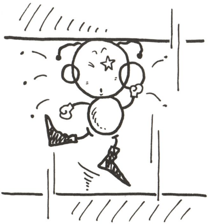

**図11.4 相対アドレスの求め方**

アセンブルを続けていって(2)までくると，ラベル LS4 のアドレスがＤ０３４Ｈであることがわかります．ここで最初の JR 命令のオペランドを決めます．このオペランドには〝PC をどれだけ動かせばよいか〟を入れます．この例では最初の JR 命令を実行するとき(JR 命令を読み出した後)，PC はＤ００２Ｈになっています．これをＤ０３４Ｈに動かすためには，

Ｄ０３４Ｈ－Ｄ００２Ｈ＝３２Ｈ

３２Ｈを足せばよいことがわかります．そこで JR 命令のオペランドのＤ００１Ｈ番地には３２Ｈを書きます．

**図 11.4** では JR 命令で後ろ(アドレスの大きい方)にジャンプしましたが，自分よりも前(アドレスの若い方)にジャンプしたい場合もあります．こういう場合は**図 11.5** のようにします．



```
D845  3E  05        | LS5:LD  A,5
  └ジャンプ先ラベルの     |
     アドレス           |
  :                 |
D870  38  □         |     JR  C,LS5
D872                |
  └JR命令が実行されるときのPC |
```

図11.5 上にジャンプ

今度はPCをD872Hから，D845Hに移さなければなりません．さきほどと同じように両アドレスの差をとると，

D845H－D872H＝－45（10進）

とマイナスの値になってしまいます．オペランドの1バイトには正の数しか書けないというのでは，このように上にジャンプするということができなくなってしまいます．そこでオペランドの8ビット（1バイト）は符号付きの7ビットにします．ですから上の－45はD3Hになります．

## マシンコードの求め方

相対アドレスジャンプには，条件ジャンプを含めると5種類あります．表11.2に相対アドレスジャンプの変換表を示します．

変換表を見ると，ニーモニックの部分には〝JR ×，e〟となっています．〝×〟は条件を入れる部分で，〝e〟というのがオペランドとなる相対アドレスです．これが表の中のマシンコードでは〝e−2〟というようになっています．

これは，ニーモニック部分での相対アドレスは，相対アドレスジャンプ命令のあるアドレスからのものであるのに対し，実際のマシンコードはその次のアドレスから計算しなければならないためです．ですからこの変換表を見る場合は注意が必要です．

実際にニーモニックでプログラムを作る場合は，オペランドにラベルを使うことが多いでしょうから，ジャンプ命令のあるアドレスからの相対アドレスeを書くことはないでしょう．ハンドアセンブルする場合には，正しく命令の次のアドレスからの相対アドレスを計算するようにしてください．

なお，本文中で変換表を参照するのはこれが最後です．もうハンドアセンブルを行う上で問題となるような点はないので，これ以後の章では，皆さん自身で付録の変換表を見るようにしてください．

| ×　　 | 無条件 | キャリー | ノン キャリー | ゼロ | ノン ゼロ | パリティ 偶数 | パリティ 奇数 | 負 | 正 | カウント |
|---|---|---|---|---|---|---|---|---|---|---|
| | | C | NC | Z | NZ | PE | PO | M | P | |
| JR ×,e | 18 e-2 | 38 e-2 | 30 e-2 | 28 e-2 | 20 e-2 | | | | | |
| DJNZ e | | | | | | | | | | 10 e-2 |

表11.2 相対ジャンプ命令の変換表

## Challenge

### ▶相対アドレスジャンプ

ここでは相対アドレスジャンプ命令を使ってプログラムを作ってみます．アセンブルするときに，オペランドの決め方に注意してください．

### テーマ　文字列を表示する

文字列というのは，キャラクタコードが連続してメモリに格納されたものをいいます．こういうプログラムを作る場合には，〝文字列の長さはどれだけあるか？″をどうやってプログラムに教えるかがポイントになります．

文字列は必ず決まったアドレスから入れてやることにすれば，文字列の先頭アドレスはわかります．けれども，それがどこまではいっているかということは，表示したい文字列の長さによって違います．

1つの方法としては，文字列とは別に文字数を与えることが考えられます．その数だけ読み出して表示するということを繰り返すわけです．しかし，**リスト11.3**では文字列の最後に00Hという文字列の終わりを示すコードを入れておくという方法をとってみます．

## リスト 11.3

```
D000 2100D8    PSTRNG: LD      HL,0D800H
D003 7E        PST1:   LD      A,(HL)
D004 B7                OR      A       ……………“CP 0”と同じ
D005 2806              JR      Z,PSTX  ……………Aレジスタが0なら
D007 CDA200            CALL    CHPUT           ループから出る
D00A 23                INC     HL ……………次の文字
D00B 18F6              JR      PST1
D00D C9        PSTX:   RET
```

リスト11.3の“OR　A”というのは“CP　0”と同じで，文字列の最後かどうかをチェックするために，Aレジスタが0かどうかを調べて結果をZフラグに入れます（この方が1バイトですむ）．このプログラムでは文字列がD800H番地から始まるということがわかっていますが，どこで終わるかはわかりません．ただひたすら文字を読み出していって，キャラクタコードが0（00H）になるまで表示を続けます．

このプログラムを実行するときは，その前に表示する文字列を決めなければなりません．これはモニタのSコマンドでD800H番地からキャラクタコードを1文字について1バイトずつ入れておきます．そして最後に00Hを忘れずに入れてください（図11.6）．

```
>SD000 ⏎
D000 FF-21 FF-00 FF-D8 FF-7E  ┐
D004 FF-B7 FF-28 FF-06 FF-CD  │ プログラムの入力
D008 FF-A2 FF-00 FF-23 FF-18  │
D00C FF-F6 FF-C9 FF-.         │
>DD000 D00F ⏎                 ┘
D000 21 00 D8 7E B7 28 06 CD  !.リ~キ(.ヘ  } プログラムの確認
D008 A2 00 23 18 F6 C9 FF FF  「.#.よノ■■

>SD800 ⏎
D800 FF-41 FF-53 FF-43 FF-49  ┐
D804 FF-49 FF-20 FF-43 FF-4F  │ 表示したい文字列のキャラクタコードをセットする
D808 FF-52 FF-50 FF-4F FF-52  │
D80C FF-41 FF-54 FF-49 FF-4F  │
D810 FF-4E FF-[00] FF-.       ┘
             └──最後に00Hを入れる
>GD000 ⏎
ASCII CORPORATION ……………実行して表示された文字列
>■
```

図11.6 リスト11.3実行例

図11.6のように，途中に２０Hを入れておくとその分だけ文字の間にスペースを入れることができます．また０１Hから１FHまでのコントロールコードを入れてみると，いろいろおもしろいことが起こります．

**▶相対アドレスジャンプの特徴**

相対アドレスジャンプ命令のオペランドに許されるのは符号付き７ビットですから，その範囲は８０Hから７FH(−128から＋127)までです(これはインデックスレジスタのディスプレースメントと同じ)．これはJR命令の次のアドレスからの距離ですから，JR命令のアドレスから見ると−126から＋129までです．そのためジャンプ先のラベルがこの範囲よりも外にあるときは，JR命令では届かなくなります(ジャンプできない)．こういう場合は絶対アドレスジャンプ命令を使うしかありません．

また条件ジャンプの種類は絶対アドレスジャンプのときよりも少なくなり，C, NC, Z, NZの4種類しかありません．したがって，P/VフラグとSフラグの状態によって相対ジャンプするということはできません．

しかしながら，相対アドレスジャンプ命令にはマシンコードのプログラムをそのまま別の場所に移せるという長所があります．たとえば**リスト**11.2のプログラム(絶対アドレスジャンプ命令を使ったプログラム)の先頭アドレスを，D２００H番地に変えたいとします．この場合，このままでは途中のKEYTS1やKEYTS2というラベルのアドレスは変わってしまいますから，JP命令のオペランドも当然書き換えなければなりません．

ところが相対アドレスジャンプ命令を使っていると，たとえジャンプ先のラベルのアドレスが変わっても，JR命令のあるアドレスも同じ分だけずれますから，ラベルまでの距離は変わりません．ですから相対アドレスジャンプ命令を含むプログラムは，それを動かしたときにオペランド部分を書き換える必要はないのです．これはプログラムを作る上で非常に便利な性質*です．

---

*このようにマシンコードのプログラムをそのまま別の場所に動かせることを，リロケータブルといいます．

# 11.4

# ループを作る

BASICの中にはFOR〜NEXTというループを作る命令がありますが，マシン語にはDJNZという命令があり，これがループを作るための命令といえます．ループを作るときは繰り返しの回数(ループカウンタ)を決めますが，DJNZ命令ではループカウンタとしてBレジスタを使います．これ以外のレジスタは使えません．

この命令もいままでのジャンプ命令のときのように，オペランドにラベルを付けます．

⑥DJNZ　LABEL

これはFOR〜NEXTでいえばNEXTにあたる命令で，これ1つで次のような処理を行います．

(1) Bレジスタから1を引く
(2) Bレジスタが0でなければLABELにジャンプし，0になったら次の命令に進む

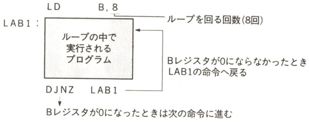



図11.7 DJNZ命令の使い方

DJNZ 命令のジャンプ先のラベルは相対アドレスで書かれますから，全体で 2 バイトの命令になります．これを使うときには図 11.7 のように，あらかじめループの回数を B レジスタに入れておきます．

ループの中で B レジスタを使うようなときには，図 11.8 のように PUSH／POP 命令を使って，B レジスタの値をスタックに保存する必要があります．

```
       LD    B,9
LAB2 : PUSH  BC  ………… ループの最初にBレジスタ(ループカウンタの値)を保存する
       ┌────────────────┐
       │ループ内の処理  │
       │この中でBレジス │
       │タの内容が壊れて│
       │もよい          │
       └────────────────┘
       POP   BC  ………… それをループの最後で取り出す
       DJNZ  LAB2        その後DJNZ命令を実行すればうまくループができる
```

図11.8 ループの中でBレジスタを使うとき

## マシンコードの求め方

DJNZ 命令は OP コードは１０Hで，その次に相対アドレスのオペランドを書きます．

DJNZ　e　……　１０　e－2

オペランド e の決め方は相対アドレスジャンプのときと同じです．

DJNZ 命令は相対アドレス指定ですから－126 から＋129 バイトの範囲にしかジャンプできません．ループの中身が大きくなって－126 バイトではとどかない場合には，図 11.9 のようにいったん下(LAB2)にジャンプして，そこから絶対アドレスジャンプで目的のラベル(LAB1)までジャンプします．B レジスタが 0 になったときは，LAB2 の下に抜けるようにするため，LAB3 というラベルも作っておきます．

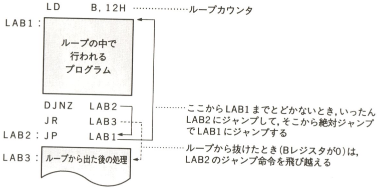



図11.9 DJNZの相対ジャンプが届かないとき

## Challenge

▶ DJNZ命令の応用

ここではDJNZ命令を使って，ループを含んだプログラムを作ってみましょう．この命令は非常に便利でよく使われますから，ぜひ使い方をマスターしてください．

### テーマ　1からNまでの足し算を行う

問題では〝1からNまで〟となっていますが，DJNZ命令を使う場合この命令の働きをよく考える必要があります．この命令はBレジスタが0かどうかによって，ループから抜けるか再びループを回るかが決まります．ですからふつうこれをループカウンタとして使うときは，最初にBレジスタに正の数を入れてループを回るたびに1つずつ減っていくようにします．

このプログラムを作るときも，N回回るループを作ればよいのですから，BレジスタにNをロードしDJNZ命令でループにします．そのループの中でBレジスタの内容を別のレジスタに加えていきます．ですからこのプログラムは〝Nから0まで〟の足し算を行うことになります．もちろん足し算ですから，足す順番が逆になっても結果は同じです．

リスト 11.4 がこのプログラムです. 1 回のループの中で, B レジスタの値を A レジスタに加えています. ただし N の値がはいる B レジスタは 8 ビットですが, 合計を計算すると 8 ビットに収まらなくなることがあります. そのため答は 16 ビットで出すことにします. ループの中では上位が C レジスタ, 下位が A レジスタで計算をして, ループから出たところでこれを HL レジスタに入れています.

最初の B レジスタのロード命令のオペランドを変えると, いろいろな N に対して合計を求めることができます.

## リスト 11.4

```
D000 0624                 LD    B,24H ·····このオペランドがNの値になる
D002 3E00                 LD    A,0   } 合計がはいる下位(Aレジスタ)と
D004 4F                   LD    C,A   } 上位(Cレジスタ)を0にしておく
D005 80        SIGMA:     ADD   A,B ···············下位8ビットを足す
D006 3001                 JR    NC,SIGMA1
D008 0C                   INC   C ······CYフラグが1なら上位8ビットに
D009 10FA      SIGMA1:    DJNZ  SIGMA                    1を足す
D00B 61                   LD    H,C } 2バイトの合計をHLレジスタに
D00C 6F                   LD    L,A } ロードする
D00D CD08E0               CALL  PUTHL ···············合計を表示する
D010 C9                   RET
```

```
>SD000 ⏎
D000 FF-06 FF-24 FF-3E FF-00 }
D004 FF-4F FF-80 FF-30 FF-01 } プログラムの入力
D008 FF-0C FF-10 FF-FA FF-61 }
D00C FF-6F FF-CD FF-08 FF-E0 }
D010 FF-C9 FF-.
>DD000 D010 ⏎
D000 06 24 3E 00 4F 80 30 01   .$>.O♠0. } プログラムの確認
D008 0C 10↑FA 61 6F CD 08 E0   ..ﾅaoﾍ.ﾀ }
D010 C9   └──この値が"N"になる  ﾉ

>GD000 ⏎ ··············実行
029A ··············答が表示される
>■
```

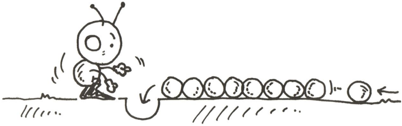
図11.10 リスト11.4実行例

## MSX-DOS

最近 MSX-DOS(エムエスエックスドス)という文字をよく見るが，これはどのようなものなのだろうか？

MSX-DOS(MSX Disk Operating System)とは，米国マイクロソフト社の開発した MSX 用のディスク・オペレーティング・システムの名前だ．ディスク・オペレーティング・システムとは，コンピュータに備わっているハードウェアや周辺装置を制御したり，ディスク上のファイルの管理を行い，ユーザーのプログラム開発を助けるものだ．具体的にいうと，ファイル名の一覧表示，ファイルのコピー，削除，ファイルの内容の表示などを，コマンドを入力するだけで実行することができる．また，DOS を使用する最も大きなメリットは，DOS の BIOS をマシン語のプログラムで利用できることにある．使い方は BASIC の BIOS とほぼ同じだが，主にディスクやファイル操作を行うルーチンがその中心となっている．そのため，BASIC の BIOS を使っただけではできないようなディスクとのファイルの入出力が容易に実現できるのだ．

DOS を使用するもう 1 つの注目すべき点は，豊富なソフトウェアを利用できることだ．まだ MSX-DOS は発表されたばかりなので，ソフトウェアが十分にそろっているとはいえないが，将来的には MS-DOS(16 ビット CPU の DOS)のように，様々なプログラミング言語やアプリケーションソフトが登場することになるだろう．



# 12章

# ビット単位のデータ操作

# 12.1

# データをビット単位で扱う命令

この章ではローテイト，シフト，ビット操作というような，データをビット単位で扱う命令を紹介します．

これまで見てきた命令というのは，メモリにしてもレジスタにしても，1バイト単位でデータを扱ってきました．しかしここで見ていく命令は，あるレジスタやメモリにあるデータの1ビットだけを扱ったり，レジスタの内容を1ビットずつずらすということを行います．

〝マシン語を使えばキメの細かい仕事ができる〞といわれているのは，ここにあげるようにビット単位でデータを扱えるからでしょう．BASICで同じような仕事をしようと思っても，1ビット単位でデータを扱う命令が用意されていないため，かえってめんどうな処理が必要になります．

## この章に出てくる命令

この章で扱う命令のうち，まず**ビット操作命令**といわれるBIT（ビット），SET（セット），RES（リセット）の3つは，あるレジスタの指定した1ビットだけを扱う命令です．この3つの命令は，よく〝他の命令の組合せで代用できる〞といわれますが，命令が1つで事足りるならば，これらの命令を使った方が便利ですから，知っておいて損はありません．

次に**ローテイト／シフト命令**というのは，レジスタの8ビットを1ビットずつ左や右にずらすという操作を行うものです．これはマシン語の演算命令でできないとされていた，掛け算を実現するのに非常に有用な命令です．またグラフィック画面でデータを扱うときなどにもよく使われます．

# 12.2

# ビット操作命令

ビット操作命令は8ビットのデータ(8ビットレジスタやHL, IX, IYレジスタで指定されるメモリの内容)の，指定したビット(桁)を1にするもの，0にするもの，いまどちらの状態であるかを調べるものの3種類があります．ニーモニックは順にSET, RES, BITで，次に示すように操作するビット番号とレジスタ名(またはメモリ)をオペランドにとります．

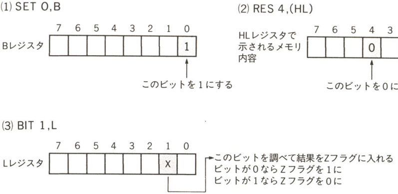



図12.1 ビット操作命令(1)〜(3)の働き

図12.1は，ビット操作命令の動作を表したものです．(1)はBレジスタのビット0を1にし(セットする)，(2)は(HL)のメモリの内容のビット4を0にします(リセットする)．(3)のBITという命令は指定したビットが0か1かを調べます．この場合Lレジスタのビット1を調べて，その結果をZフラグに入れます．Zフラグですから，その調べたビットが0のときに1になり(セットされ)，ビットが1のときはリセットされて0になります．なお，SET命令とRES命令はフラグには影響を与えません．

# 12.3

# ローテイト／シフト命令

ローテイト／シフト命令は，8ビットのデータを右や左に回したり，ずらしたりする命令です．〝ビットをずらす〟というのは図12.2のように，あるビットを左隣りか右隣りのビットに移動するということを，8つのビットすべてに対して行うことです．

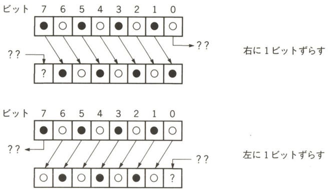



図12.2 ビットをずらす

## ローテイト命令

ローテイト命令はビットを〝回転〟させる命令です．回転というのは，ビットをずらしたときに端からはみ出したビットを，反対側から入れることです．ローテイト命令には〝ローテイト〟と〝ローテイトサーキュラ〟という2種類があって，それぞれ右と左に回転させることができます．

まずローテイトサーキュラ命令は，図12.3の(1)と(2)です．RLC(ローテイト・レフト・サーキュラ)命令は，ビット0をビット1に，ビット1をビット2に……というよ

うにずらしていき，ビット7をビット0とCYフラグに入れる命令です．RRC(ローテイト・ライト・サーキュラ)命令は逆に右にずらしていって，ビット0をビット7とCYフラグに入れます．

ローテイト命令は(3)と(4)のように，CYフラグをビット0とビット7の間に入れて，9つのビットで回転を行うものです．

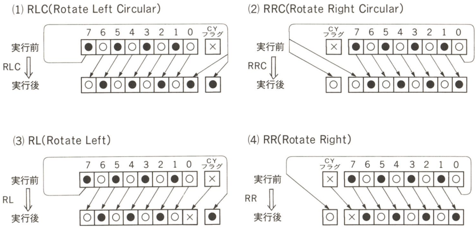



**図12.3 ローテート命令**

ローテイト命令のニーモニックは次に示すように図12.3のOPコードのそれぞれに，レジスタまたは(HL)，(IX+d)，(IY+d)をオペランドに付けたものです．

```
① RLC  B
② RRC  H
③ RL   (HL)
④ RR   A
```

ただしAレジスタだけには，〝RLC　A〟，〝RR　A〟のほかに，〝RLCA〟，〝RRA〟などのように，オペランドの〝A〟をOPコードにくっつけた形のものがあります．これはほとんど同じ働きをするものですが，フラグに与える影響が少し異なっています．

## ローテイト命令の働き

ここで実際にローテイト命令はどのように働くかを見てみましょう．Aレジスタに49Hがあるときに〝RRC　A″を実行すると図12.4のようになります．

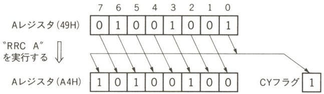



図12.4 RRC A命令

AレジスタはC4Hになり，ビット0にあった1がCYフラグにはいります．

またBレジスタが8EH,CYフラグが0の場合に〝RL　B″を実行すると，図12.5のようになります．

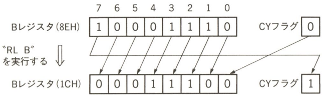



図12.5 RL B命令

## シフト命令

シフト命令は，ビットを左右にずらすという点はローテイト命令と変わりません．ただ左にシフトしたときのビット0，右にシフトしたときのビット7にはいってくるものが違います．

シフトにも命令が2種類あって，SL(シフトロジカル)とSA(シフトアリスメティック)に分けられます．アリスメティックにはL(左)とR(右)がありますが，ロジカル

の方にはSRL命令つまり右にシフトする命令しかありません．

まずSLA命令は，図12.6の(1)のように左にシフトして，ビット0には0がはいり，ビット7にあったものはCYフラグにはいります．

右にシフトする場合，ロジカルとアリスメティックではビット7にはいるものが違います．SRL命令では(2)のように0がはいりますが，SRA命令だと前のビット7の値がそのまま残ります．なお，どの命令もCYフラグにははみ出したビットの値がはいります．

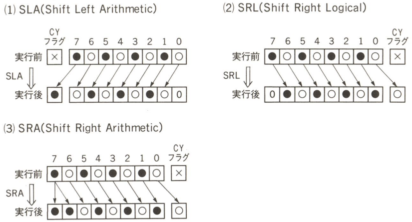
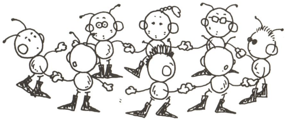



図12.6 シフト命令

ニーモニックはローテイト命令の場合と同じく，OPコードには8ビットのレジスタか(HL)，(IX+d)，(IY+d)を書きます．ただしローテイト命令にあったような〝RRA″というようにオペランドの〝A″がOPコードにくっついた形のものはありません．

①SLA　A
②SRL　(HL)
③SRA　C



## シフト命令の応用

左シフトには〝2倍にする〟という意味があります．SLA 命令を行うと，2進数の桁を1つずつ左にずらして，1の位に0を入れたということになります．つまり桁が1つ上がったことになるのです．

たとえばA＝0 3 Hのときに〝SLA　A〟を実行すると図12.7のように3が2倍の0 6 Hになることがわかります．

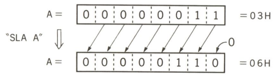



図12.7 左シフトしてレジスタの値を2倍にする

Z80では掛け算命令がありませんが，SLA 命令1回で2倍，2回で4倍，3回で8倍というように，2の乗数倍は簡単に実現できます．

逆に右シフトをするということは，値を半分にするという意味になります．たとえば図12.8を見てください．

A＝F E Hを254と考えると，SRL 命令によって7 F Hつまり127が得られるので

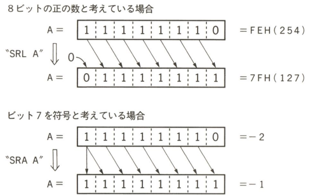



図12.8 右シフトしてレジスタの値を半分にする

ちょうど半分になっています．ところがビット7を符号と見て，−2と考えると，半分なら−1になってもらわないと困ります．そんなときに使うのがSRA命令です．つまりSRA命令で，〝ビット7をそのままにする〟というのは，符号はそのままで残すという意味があります．

## Challenge

▶ 2進数表示プログラム

ローテイト／シフト命令の結果は，これまでのChallengeのように16進数で表示してもよいのですが，やはり各ビットごとに0か1かを表示する2進数の方がわかりやすいでしょう．そこでChallengeの前にAレジスタの内容を2進数表示する〝PUTBIN〟というルーチンを作っておきます．

このプログラムの考え方は前に作った〝PUTA(16進数表示)〟と同じです．今回はAレジスタの内容を上から1ビットずつ取っていって，1なら〝1〟，0なら〝0〟のキャラクタコードをAレジスタに入れて，BIOSのCHPUTをコールすればよいのです．ただし，このプログラムの中にはPUTCRというサブルーチンが含まれています．これは復帰改行を行うためのものです．PUTBINはAレジスタの内容を表示した後，

**リスト 12.1**

```
E040 C5       PUTBIN: PUSH  BC
E041 F5               PUSH  AF
E042 4F               LD    C,A     …表示する値をCレジスタに入れる
E043 0608             LD    B,8     …8ビット分
E045 AF       PUTA:   XOR   A       …Aレジスタは0にしておく("LD A,0"と同じ)
E046 CB21             SLA   C       …Cから1ビット取ってCYフラグに入れる
E048 CE30             ADC   A,30H   …CYフラグが1なら31H(1)，0なら30H(0)を表示する
E04A CDA200           CALL  CHPUT
E04D 10F6             DJNZ  PUTA
E04F F1               POP   AF
E050 C1               POP   BC
E051 C9               RET
                                    (E040〜E051: Aレジスタを2進数表示する)
E052 F5       PUTCR:  PUSH  AF
E053 3E0D             LD    A,0DH
E055 CDA200           CALL  CHPUT
E058 3E0A             LD    A,0AH
E05A CDA200           CALL  CHPUT
E05D F1               POP   AF
E05E C9               RET
                                    (E052〜E05E: Aレジスタを保存したまま改行するサブルーチン．キャリッジリターン(0DH)とラインフィード(0AH)を出力する)
```

カーソルの位置を変更しないので，PUTCR をコールすることによってカーソルを次の行の先頭に持ってくることができます．

このPUTBIN はＥ０４０Ｈ番地から始まるようにしています．７章で作った**リスト7.7**とははいるアドレスが違いますから，同時にメモリに入れておいて好きな方をコールすることができます．このプログラムでは，まずＡレジスタにはいってきた値をＣレジスタに入れ，ここから１ビットずつ取り出すところでSLA 命令を使います．その結果によってＡレジスタに３０Ｈ(0)か３１Ｈ(1)を入れて，BIOS の CHPUT をコールするということを，８回繰り返しています．

モニタから直接〝ＧＥ０４０〟で実行してみて，８桁の２進数が表示されればおそらく大丈夫でしょう．念のため，別のアドレスに次の**リスト12.2**を入力して試してみてください．これでＡ４Ｈの２進数表示の〝10100100〟が出れば成功です．

## リスト 12.2

```
D000 3EA4        LD      A,0A4H  ……2進数表示したい値をAレジ
D002 CD40E0      CALL    PUTBIN       スタにロードする
D005 C9          RET
```

このPUTBIN がうまくできたら準備完了です．この後の Challenge では，ローテイト／シフト命令は，それぞれどのような働きをするかを表示するプログラムを作っていきます．これらの命令を実行する前後でこの２進数表示サブルーチンをコールして，どのように変わったかを見てみます．

このサブルーチンはＡレジスタの内容を表示するように作りましたから，これから見ていく各種ローテイト／シフト命令はすべてＡレジスタを使うことにします．他のレジスタを使った場合も原理的にはまったく同じです．もし他のレジスタを試してみたい場合は，命令の前後にＡレジスタにロードして PUTBIN をコールすれば，２進数で表示することができます．

### ▶命令の動作を目で確かめる

まずはローテイト命令のうちCY フラグを使わないサーキュラ命令を使ってみます．**リスト12.3**がそのプログラムですが，RLC 命令の実行前と実行後の違いを２進数で表示します．レジスタに適当な値をロードして，それを表示した後，RLC 命令を実行します．

## リスト 12.3

```
D000 3E94      LD     A,94H   ……………初期値をAレジスタにロードする
D002 CD40E0    CALL   PUTBIN  } 初期値を表示して改行する
D005 CD52E0    CALL   PUTCR   }
D008 CB07      RLC    A       ……………ローテイト命令の実行
D00A CD40E0    CALL   PUTBIN  ……………ローテイト命令実行後の値を表示する
D00D C9        RET
```

これを入力して実行してみたのが図 12.9 です．このように最初〝10010100″次に〝00101001″が表示されれば成功です．

```
>SD000 ⏎
D000 FF-3E FF-94 FF-CD FF-40   }
D004 FF-E0 FF-CD FF-52 FF-E0   } プログラムの入力
D008 FF-CB FF-07 FF-CD FF-40   }
D00C FF-E0 FF-C9 FF-.          }
>DD000 D00F ⏎
D000 3E 94 CD 40 E0 CD 52 E0   >えへ@たへRた  } プログラムの確認
D008 CB 07 CD 40 E0 C9 FF FF   ヒ.へ@たノ■■   }

>GD000 ⏎ ……………実行
10010100 ……………初期値(94H)
00101001 ……………変換後の値(29H)
>■
```

図12.9 リスト12.3実行例

CY フラグの値がかかわってくる場合には，ローテイト命令を行う前にこのフラグに 0 か 1 かの値を書き込んでおきます．CY フラグを 0 にするには〝OR　A″，1 にするには SCF(Set Carry Flag )という命令を使うと便利です(リスト 12.4, 12.5).

## リスト 12.4

```
D000 3E83      LD     A,83H
D002 CD40E0    CALL   PUTBIN  } 初期値を表示して改行する
D005 CD52E0    CALL   PUTCR   }
D008 37        SCF            ……………CYフラグを1にする
D009 CB1F      RR     A
D00B CD40E0    CALL   PUTBIN  ……………RR命令実行後の値を表示する
D00E C9        RET
```

```
>SD000 ⏎
D000 FF-3E FF-83 FF-CD FF-40
D004 FF-E0 FF-CD FF-52 FF-E0     プログラムの入力
D008 FF-37 FF-CB FF-1F FF-CD
D00C FF-40 FF-E0 FF-C9 FF-.
>DD000 D00F ⏎
D000 3E 83 CD 40 E0 CD 52 E0   >◆へ@たへRた   プログラムの確認
D008 37 CB 1F CD 40 E0 C9 FF   7ヒ.へ@たノ■

>GD000 ⏎ ……………実行
10000011 ……………初期値(83H)
11000001 ……………変換後(C1H)
>■
  CYフラグからはいってきたビット
```

図12.10 リスト12.4実行例

## リスト 12.5

```
D000 3E45      LD    A,45H
D002 CD40E0    CALL  PUTBIN
D005 CD52E0    CALL  PUTCR
D008 CB2F      SRA   A  ………ビット7の値を変えずに
D00A CD40E0    CALL  PUTBIN       右シフトする
D00D C9        RET
```

```
>SD000 ⏎
D000 FF-3E FF-45 FF-CD FF-40
D004 FF-E0 FF-CD FF-52 FF-E0     プログラムの入力
D008 FF-CB FF-2F FF-CD FF-40
D00C FF-E0 FF-C9 FF-.
>DD000 D00F ⏎
D000 3E 45 CD 40 E0 CD 52 E0   >Eへ@たへRた   プログラムの確認
D008 CB 2F CD 40 E0 C9 FF FF   ヒ/へ@たノ■■

>GD000 ⏎ ……………実行
01000101 ……………初期値(45H)
00100010 ……………変換後の値(22H)
>■
```

図12.11 リスト12.5実行例

# 12.4
# シフトを使った掛け算

シフト命令の応用として，掛け算の実現があります．Z80 には掛け算をするための命令がありません．ですから掛け算が必要になったら，そのためのプログラムを作ってやる必要があります．

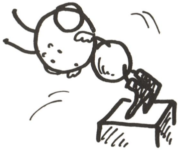
この節で紹介する掛け算プログラムは，8 ビットの数と 8 ビットの数を掛けて 16 ビットの答(積)を得るというものです．これは Z80 を利用して，掛け算のプログラムを作る場合に，最もよく使われる手法で，〝定石〟ともいえるものです．

## 掛け算の問題点

足し算や引き算の場合には，8 ビットと 8 ビットで演算を行うと答も 8 ビット(CY フラグを考えると 9 ビット)になります．しかし掛け算を行ったときは，桁が大きくなり，8 ビット×8 ビットの答は 16 ビットになります．

これは 10 進数 2 桁どうしの掛け算を考えてみればわかります．10×20 は 200 となりますが，99×99 は 4 桁の数 9801 になりますから，結局 2 桁×2 桁なら 4 桁以内に収まります．一般に N 桁と M 桁の掛け算をすれば，その答は高々(N＋M)桁です．

ですから掛け算のプログラムを作ろうと思ったら，このことを考えておく必要があります．8 ビットに 8 ビットのデータを掛けてその結果を 8 ビットのレジスタに収めるということはできません．答は 16 ビットレジスタに入れるようにしなければなりません．



## プログラムを作る

さて掛け算のプログラムを作るときにまず思いつく方法は，掛けられる数を掛ける数の回数だけ足すというものです．これなら足し算という命令を使えるし，〝掛ける数の回数〃というのも，DJNZ 命令を使ってループを作ればできそうです．

しかし，これではあまり工夫がありません．この方法では掛ける数が大きくなると，それだけループを回る数が増えますから実行時間もよけいにかかってしまいます．そこでもっとエレガントな方法を使います(**リスト** 12.6)．

### リスト 12.6

```
MULT:   PUSH  BC
        PUSH  DE
        LD    D,0
        LD    L,0
        LD    B,8
MULT1:  ADD   HL,HL ……………HLレジスタの内容を左にずらし(HLレジスタを2倍し)，ビット7をCYフラグに入れる
        JR    NC,MULT2 ……………CYフラグが0のときはMULT2へジャンプする
        ADD   HL,DE ……………CYフラグが1のときはEレジスタの内容をHLレジスタに足す
MULT2:  DJNZ  MULT1
        POP   DE
        POP   BC
        RET
```

このプログラム(サブルーチン)はＥレジスタとＨレジスタにはいっている値を掛けて，その結果を HL レジスタに入れるというものです．どうしてこんな短いプログラムでできるのか不思議に思えますが，とりあえず使ってみるとうまくいくのです．

これにサブルーチンをコールするメインルーチンを付け加えてみましょう．**リスト** 12.7 は E, H レジスタに値をロードして掛け算ルーチンをコールし，結果を表示するプログラムです．

**リスト** 12.7 の実行結果が**図** 12.12 です．８７Ｈ×２９Ｈの答，〝１５９Ｆ〃がプリントされるはずです．〝そんなバカな〃と思う人は，メインルーチンを書き換えていろいろな値の掛け算を試してみてください．

## リスト 12.7

```
D000 1E87                 LD     E,87H  } データをロードする
D002 2629                 LD     H,29H  }
D004 CD0BD0               CALL   MULT  ……………掛け算
D007 CD08E0               CALL   PUTHL ……………答を表示する
D00A C9                   RET
                          ;
D00B C5         MULT:     PUSH   BC
D00C D5                   PUSH   DE
D00D 1600                 LD     D,0
D00F 2E00                 LD     L,0
D011 0608                 LD     B,8
D013 29         MULT1:    ADD    HL,HL
D014 3001                 JR     NC,MULT2
D016 19                   ADD    HL,DE
D017 10FA       MULT2:    DJNZ   MULT1
D019 D1                   POP    DE
D01A C1                   POP    BC
D01B C9                   RET
```

```
>SD000
D000 FF-1E FF-87 FF-26 FF-29  }
D004 FF-CD FF-0B FF-D0 FF-CD  }
D008 FF-08 FF-E0 FF-C9 FF-C5  }
D00C FF-D5 FF-16 FF-00 FF-2E  } プログラムの入力
D010 FF-00 FF-06 FF-08 FF-29  }
D014 FF-30 FF-01 FF-19 FF-10  }
D018 FF-FA FF-D1 FF-C1 FF-C9  }
D01C FF-.
>DD000 D01F ⏎
D000 1E 87 26 29 CD 0B D0 CD  .ぁ&)ヘ.ミヘ  }
D008 08 E0 C9 C5 D5 16 00 2E  .たノナユ...  } プログラムの確認
D010 00 06 08 29 30 01 19 10  ...)0...  }
D018 FA D1 C1 C9 FF FF FF FF  れムチノ■■■■  }

>GD000 ⏎ ……………実行
159F ……………掛け算(87H×29H)の答
>■
```

図12.12 リスト12.7実行例

## 掛け算の原理

このプログラムを理解するためには，〝シフトによる掛け算〟を理解しなければなりません．たとえば**リスト** 12.7 の８７Ｈ×２９Ｈという掛け算を考えてみましょう．CPU の中では２進数で演算が行われるので，これは次のように書き換えることができます．

１０００１１１×００１０１００１

これをふつう私達がやるような筆算で計算すると**図** 12.13 のようになります．

```
          1 0 0 0 0 1 1 1     掛けられる数 87H
      ×)  0 0 1 0 1 0 0 1     掛ける数     29H
      ---------------------
          1 0 0 0 0 1 1 1
    1 0 0 0 0 1 1 1
1 0 0 0 0 1 1 1
---------------------------
1 0 1 0 1 1 0 0 1 1 1 1 1    →159FH(答)
```

**図12.13　掛け算**

掛ける数の各ビットを見て，０ならば何もせず，１ならば掛けられる数をその桁のところまでずらしておき，後でそれをすべて加えます．10 進数で行うのと原理的には何ら変わりませんが，２進数で行うと各桁に０か１しか出てこないので，１なら掛けられる数を足して，０なら足さないというように，各桁では掛け算をする必要はありません．

**リスト** 12.6 のプログラムでは，この原理を使っています．つまり掛ける数のビットを１つずつ見ていき，それが１ならば掛けられる数を足し，０ならば何もしないということを行っています．

ただし，足していく順番が上の桁から行うのと，掛けられる数と答が同じレジスタ(HL レジスタ)を共有しているというところがポイントです．この**リスト** 12.6 の作業を表したのが**図** 12.14 です．

```
    10000111        Eレジスタ(掛けられる数)
×)  00101001        Hレジスタ(掛ける数)
      10000111        ┐
         10000111     │ 部分積
                      │ Hレジスタのビットが1の桁のところに
            10000111  ┘ Eレジスタを持ってくる
      1010110011111   …答　部分積をすべて加える
```

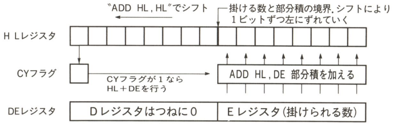
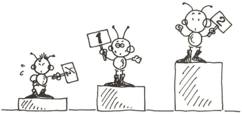



図12.14 リスト12.6の掛け算のしくみ

図12.14のうちHLレジスタの太線になっているところが，HLレジスタのうち"掛ける数"と"答"の境になっています．この境界線はシフトを行うたびに左に動き，これが8回のシフトでHレジスタからすべて出てしまったとき，HLレジスタが答になっています．

"ADD　HL，HL"という命令がポイントで，この命令は答を1ビット左にシフトするというのと，Hレジスタのビット7をCYフラグに入れるという2つのことを同時に行います．CYフラグが1ならば，掛ける数の桁が1であったことになりますから，Eレジスタの内容をHLレジスタに加えます．

この動作を8回(Hレジスタの大きさ)だけ繰り返せば，HLレジスタに答がはいります．



## プログラムのスピードアップ

マシン語のプログラムは，BASICで書いたものよりはるかに高速だが，ときにはシビアなスピードが要求されることがある．こんなときは，まずアルゴリズムを考え直す必要がある．たとえば，バイト単位の掛け算をする場合，シフトと足し算の繰り返しで実行するのと，単に足し算を繰り返すのでは結果の違いは明らかだ．

また，同じアルゴリズムでもプログラムの書き方によってスピードに差が出ることもある．一般にレジスタ-レジスタ間の転送の方が，メモリ-レジスタ間の転送よりも速いので，次のような例では，②の方が速くなる．

①
```
PUSH BC
POP  DE
```

②
```
LD   D,B
LD   E,C
```

スピードアップをはかる思い切った方法としては，ループやサブルーチンを展開することがある．たとえば，シフトを3回行うループは③のようになるが，これを④のように書き換えてしまうのだ．

③
```
      LD   B,3
LOOP: SLA  A
      DJNZ LOOP
```

④
```
SLA  A
SLA  A
SLA  A
```

この例では大きな違いはないが，こういったループを何万回も通る場合には，かなり効果的な方法だ．

# 13章

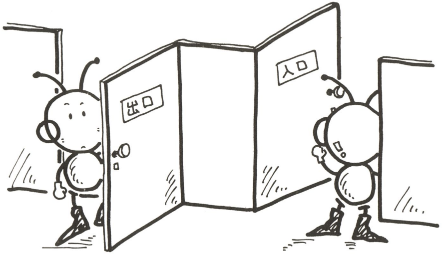
# I/Oポートの利用

# 13.1

# I／Oポートを使う

CPUにとっては，自分とメモリだけが〝自分の世界〟です．プログラムはメモリから読み出してきますし，データも通常メモリに置いてアクセスします．しかしこれだけでは，外部とデータをやり取りすることすらできません．そこで必要になるのが，このI／Oポートなのです．

CPUにとっての〝外部〟とは，プリンタやディスクのような周辺装置はもちろん，PSGとかVDPなどマシン本体にはいっている他のLSIも含まれています．I／Oポートを扱う命令を**入出力命令**といい，CPUがI／Oポートに対してデータを送り出すことを出力，I／Oポートからデータを取ってくることを入力といいます．



## MSXのI/Oポート

同じZ80というCPUを使っていても，0番から255番まであるI/Oポートをそれぞれ何に割り当てているかというのは，コンピュータによって違います．ですからI/Oポートを使うには，入出力命令のほかに，何番のポートは何のために使われているかという知識が必要になります．

MSXはどうなっているかというと，256個あるポートのうち，規格(すべてのMSXに共通のもの)として決まっているのは次のものです．

- ○ RS-232 C*
- ○ プリンタ
- ○ VDP(Video Display Processor：画面を扱う)
- ○ PSG(Programmable Sound Generator：音を出すもの)
- ○ PPI**(Programmable Peripheral Interface)

MSXの規格では，上のもの以外のI/Oポートについては互換性***を保証していません．ですからこの本では，上にあげた〝規格内〟のI/Oポートを使うことにします．それぞれの詳しい構造は，後で命令を実際に使うところで説明します．

MSXではこれらI/Oポートを扱うために，BIOSルーチンというものがあります．ですから本当は，入出力命令を使わなくても，同じことがBIOSルーチンをコールすることによって実現できます．

MSX用に売られているソフトウェアでは，入出力命令を使ってI/Oポートを操作することは禁じられており，必ずBIOSルーチンを使うことになっています．これは，将来I/Oポートの規格が変更になるかも知れないからです．

この章では，入出力命令を使ってサンプルプログラムを作りますが，同様のことが，BIOSルーチンを使えばより簡単にできるということを，頭に入れて読んでください．

---

*　外部との通信用インターフェイス

**　スロット，キーボード，カセットなどを扱うLSI

*** MSX全機種に共通であること

# 13.2
# 入出力命令

I/Oポートは全部で256個あります．ですから入出力命令を実行するには，〝どのポートか〟ということを指定するために，8ビットのポート番号が必要になります．

## ニーモニックの書き方

入出力命令は，Aレジスタで行います．入力するときには指定したポートから読んだデータがAレジスタにはいり，出力の場合は出力したいデータをAレジスタに入れて，それを指定したポートに出力します．この2つの命令のニーモニックはそれぞれ①と②です．

```
① IN    A,(89H)
② OUT   (91H),A
```

①は89H番ポートからの入力，②は91H番ポートへの出力です．オペランドはちょうどロード命令のように，デスティネーション(送り先)とソース(送り元)がこの順に並んでいるわけです．

また，③，④で示すように，Cレジスタの値をポート番号にして入出力を行うこともできます．この場合は8ビットのレジスタすべてで入出力が可能です．

```
③ IN    B (C)
④ OUT   (C),L
```

③を実行すると，Cレジスタの内容が89Hなら89H番ポートから入力したデータがBレジスタにはいります．④は同様に，LレジスタのデータをCレジスタで示される番号のポートに出力します．

この命令は同じような機能のポートがいくつも並んでいて，その中からポート番号を計算して入出力を行う場合とか，使用するポート番号がメモリのどこかにデータとして置いてあるため，①や②のようにプログラムのオペランドに，直接ポート番号を書くことができないときに使われます．

ここでポートについて1つ注意があります．CPUと外部機器(他のLSIも含む)は通常入力も出力も，同じポート番号で行われます．しかし相手の機械の性質によって入力にしか使われないものとか，出力にしか使われないものがあります．たとえばプリンタのようなものは出力専用ですし，キーボードは入力にしか使いません．

もちろん両方に使えるものもありますが，入力と出力では意味が違ったりすることがよくあります．ですからメモリのように，同じポート番号に出力した(OUTした)値が，次に入力した(INした)ときに残っているとは限りません．この点メモリとはまったく異なっています．

## I/Oポートの構造

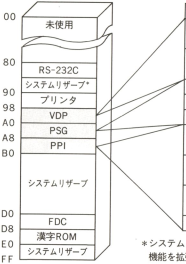
入出力命令を使うには，まずI/Oポートの構造についてよく知らなければなりません．そこでまずMSXのI/Oポートがどのような規格になっているかを調べてみましょう．

MSXではっきりとポート番号が割り当てられているのは，RS-232C，プリンタ，VDP, PSGそれにPPIです．表13.1からわかるように，各装置には入力と出力で同じポート番号を割り当てているところがほとんどです．

VDPは，画面に出力する情報をデータとして送り出すところですが，そのためにはこのVDPというLSIの複雑な構造を理解する必要があります．またRS-232CというのはMSXを他のコンピュータとつないだときに，通信用に使用するものです．



R：入力　W：出力




| ポート番号 | R/W | 内　　容 |
|---|---|---|
| 98 | W<br>R | V-RAMへのデータライト<br>V-RAMからのデータリード |
| 99 | W<br>R | コマンド，アドレスセット<br>ステータスリード |
| A0 | W | アドレスラッチ |
| A1 | W | データライト |
| A2 | R | データリード |
| A8 | W<br>R | ポートAデータライト<br>ポートAデータリード |
| A9 | W<br>R | ポートBデータライト<br>ポートBデータリード |
| AA | W<br>R | ポートCデータライト<br>ポートCデータリード |
| AB | W | モードセット |

＊システムリザーブと書かれているI/Oポートは,将来MSX-BASICが機能を拡張するためにとってあるところなので，ユーザーが使ってはならない

表13.1　I/Oマップ

## Challenge

### ▶音の出し方

BASICでPSGを扱う命令といえばSOUND文があります．これは次のように書きます．

```
SOUND 7,&H97
```

PSGの中にもレジスタが14個あります．上の命令は，PSGのレジスタ番号と，そこに書き込みたいデータを指定してPSGのレジスタに書き込みを行うものです．

マシン語でプログラムを作る場合もまったく同様に，PSGのレジスタとデータを指定して書き込みを行います．PSGのレジスタは14個ですから，本当はそれぞれ別々に14個の出力ポートがあればよいのですが，そうはなっていません．

PSGの場合，まずＡ０Ｈ番ポートにレジスタ番号を出力し，その後Ａ１Ｈ番ポートにデータを出力します．そうすると，後はPSGの方でうまくそのレジスタに書き込んでくれます．

たとえばさきほどのSOUND文は,

```
OUT  &HA0,7
OUT  &HA1,&H97
```

とすればよいのです．OUT文はBASICでポートに出力を行う命令です．このプログラムをマシン語では次のように書きます．

```
LD     A,7
OUT    (0A0H),A
LD     A,97H
OUT    (0A1H),A
```

このように，まずポートに出力するデータをAレジスタに入れ，次にポート番号を指定して出力します．この場合，必ず〝レジスタ→データ〟という順番を守ってください．先にデータをA1H番ポートに出力してしまうと，それがどのレジスタに書かれるかわかりません．

ところでPSG用のA2H番ポートは〝データリード〟と書いてあります．これはPSGのレジスタの値を読んでくるポートです．これを使うには，まずA0H番にレジスタ番号を出力しておきます．次にこのA2H番ポートからデータを入力すると，その番号のレジスタの内容がわかるのです．このようにPSGのレジスタの内容を読み出すというのは，BASICにはない機能です．特に7番レジスタの内容がわかると，いまどの音が出ているかということがわかりますから，結構使う場面があるかもしれません．

## テーマ　PSGのレジスタにデータを書き込んで音を出す

**リスト**13.1を見ると，レジスタに書き込む値というのが，プログラムとは別にデータとしてメモリにはいっているのがわかります．そのデータのあるメモリのアドレスをHLレジスタで指定して，そこからデータを1つずつ取り出してはA1H番ポート出力するということを14回のループで行っています．

BASICの場合には〝SOUND　0，100〟というように1つの命令ですむところが，

## リスト 13.1

```
D000 2100D8           LD    HL,0D800H  …データのあるアドレス
D003 060E             LD    B,14       …14回ループを回す
D005 3E0E     PSG1:   LD    A,14     } レジスタ番号．Bレジスタが14のとき0，
D007 90               SUB   B        } 13のとき1，…，1のとき13がAレジスタにはいる
D008 D3A0             OUT   (0A0H),A   …レジスタ番号を出力する
D00A 7E               LD    A,(HL)     …データをメモリから読み込む
D00B D3A1             OUT   (0A1H),A   …データを出力する
D00D 23               INC   HL         …次のデータのあるアドレス
D00E 10F5             DJNZ  PSG1
D010 C9               RET
```

```
>SD800 ⏎
D800 00-1E 00-__ 00-__ 00-       } PSGのレジスタ0番から13番までに書き込む
D804 00-__ 00-__ 00-__ 00-3E     } データを入力する
D808 00-10 00-__ 00-__ 00-67
D80C 00-5E 00-08 00-.__
>GD000 ⏎…………実行

>■          (音を止めたいときは，いったんストップをかけてモニタを終了してください)
```

図13.1 リスト13.1プログラムの確認と実行

マシン語だと前に出てきたように4つも命令が必要になります．ですからこの4つの命令を必要なレジスタの数だけ並べるというよりも，0番から13番までのすべてのレジスタについてループの中で値を書いてしまった方が，プログラムは短くてすみます．

なかには書き込む必要のないレジスタもありますが，そういうものには適当な値を書いておいて，7番レジスタに書き込むデータの値でその音を出さないようにします．このようにレジスタに書き込みたい値をデータとしてまとめた方が便利です．

### ▶PPIのコントロール

次にPPIを使って〝どのキーが押されているか？〞を調べるプログラムを作ってみましょう．PPIというのは，CPUに対するいろいろな入出力をコントロールするLSIで，表13.2に示すようにスロットとかカセットテープなども扱いますが，ここではキーボードについて見ていきます．

| ポート | ビット | 内　　　容 |
|---|---|---|
| A<br>(A8H番)<br>出　力 | 0<br>1 | 0000H～3FFFH番地のスロット指定信号 |
| | 2<br>3 | 4000H～7FFFH番地のスロット指定信号 |
| | 4<br>5 | 8000H～BFFFH番土のスロット指定信号 |
| | 6<br>7 | C000H～FFFFH番地のスロット指定信号 |
| B<br>(A9H番)<br>入　力 | 0<br>～<br>7 | キーボードリターン信号 |
| C<br>(AAH番)<br>出　力 | 0<br>1<br>2<br>3 | キーボードスキャン信号 |
| | 4 | カセットコントロール(0でON) |
| | 5 | カセット書き込み信号 |
| | 6 | CAPSランプ信号(0で点燈) |
| | 7 | ソフトによるサウンド出力 |

**表13.2 PPIビット割り当て**

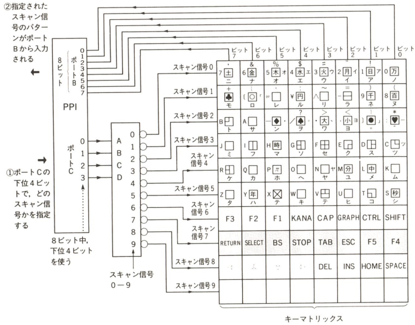



**図13.2 キーボード配列**

PPIにはA８HからAAHまでのポート番号が割り当てられていて，それぞれA, B, Cというポートになっています．このうちポートB(A９H番)からデータを入力すると，キーボードのそれぞれのキーが押されているかいないかという状態がわかります．しかしこれを入力する場合には，その前にポートC(AAH番)に**キーボードスキャン信号***を出さなければなりません．

キーボードは図13.2のように，8つずつ組みになっています．ポートCに出力するデータの下位4ビットで〝スキャン信号〞を指定すると，このうちの1組が選択され，次にポートBから1バイト入力するとその8つのキーの状態がわかるというしくみです．

たとえばリターンキーは，スキャン信号が7番ですからポートCに０７Hを出力してから，ポートBから入力してきたときそのビット7を見ればわかるわけです．各キーはそれが押されていれば対応するビットが0，押されていなければ1になります．

ここでちょっとBASICで実験してみましょう．次の**リスト13.2**を実行してみてください．

## リスト 13.2

```
100 OUT &HAA,7  ……………キーボードスキャン信号をポートCに出力する
110 X=INP(&HA9) ……………ポートBから入力する
120 PRINT RIGHT$("0000000"+BIN$(X),8) ……………2進数で表示する
130 GOTO 100
```

このプログラムを走らせると，〝11111111〞という数字が際限なく表示されます．ここで図13.2で黒く塗られているキーを押してみてください．図13.2のいちばん上に各キーに対応するビットが示してありますが，押したキーのビットが0になるのがわかります．

マシン語のプログラムでは，キーボードすべてについて，それぞれが押されているかいないかを表示するというものを作ってみます．**リスト13.3**では12章で作ったPUTBINというサブルーチンを使用しています．

---

＊どのキーの組みかを指定する信号

**テーマ** キースキャンの状態を画面に表示する

## リスト 13.3

```
D000 210201    SCAN:   LD      HL,0102H  } カーソルの初期位置を
D003 CDC600            CALL    POSIT     } 設定する(1,2)
D006 0609              LD      B,9
D008 3E09      SCAN1:  LD      A,9
D00A 90                SUB     B ……………スキャン信号を得る
D00B D3AA              OUT     (0AAH),A
D00D DBA9              IN      A,(0A9H) ……キーボードのデータを得る
D00F CD40E0            CALL    PUTBIN   ……文字を表示する
D012 3E0D              LD      A,0DH    ]
D014 CDA200            CALL    CHPUT    ] 改行して次のデータを表示する
D017 3E0A              LD      A,0AH    ]
D019 CDA200            CALL    CHPUT    ]
D01C 10EA              DJNZ    SCAN1
D01E CDB700            CALL    BREAKX   } CTRL+STOP が押されたら
D021 D8                RET     C        } BASICに戻る
D022 18DC              JR      SCAN
```

〔注〕このプログラムはMSXの標準システムのキーだけをスキャンします．それ以外のキーがあるものについては見ていません．

図13.3がこのプログラムの実行例ですが，MSXのすべてのキーについてそれが押されているかいないかを，0か1かで表示しています．ここでどれかキーを押してみると，そのキーに対応するビットが0になるのがわかります．

このプログラムの特徴は，一度に2つ以上のキーを押してもそれがわかるということです．試しにどれでも2つ以上のキーを押してみてください．ちゃんとその数に対応して0になるはずです．

押されたキーを調べるには，CHGETというBIOSルーチンがあります．これは押されたキーのキャラクタコードをAレジスタに返すというものですが，一度に2つ以上のキャラクタが押されたときは，それを感知することができません．ところがこのようにPPIを直接調べると，ここで見たように複数のキャラクタが同時に押されていても，それがわかります．

```
>SD000 ⏎
D000 FF-21 FF-02 FF-01 FF-CD
D004 FF-C6 FF-00 FF-06 FF-09
D008 FF-3E FF-09 FF-90 FF-D3
D00C FF-AA FF-DB FF-A9 FF-CD
D010 FF-40 FF-E0 FF-3E FF-0D      プログラムの入力
D014 FF-CD FF-A2 FF-00 FF-3E
D018 FF-0A FF-CD FF-A2 FF-00
D01C FF-10 FF-EA FF-CD FF-B7
D020 FF-00 FF-D8 FF-18 FF-DC
D024 FF-.
>DD000 D023 ⏎
D000 21 02 01 CD C6 00 06 09   !..ヘニ...
D008 3E 09 90 D3 AA DB A9 CD   >. モェロゥヘ
D010 40 E0 3E 0D CD A2 00 3E   @た>.ヘ「.>    プログラムの確認
D018 0A CD A2 00 10 EA CD B7   .ヘ「..はヘキ
D020 00 D8 18 DC               .リ.ワ
--------------------------------------------------------------
11111111
11111111
11111111
01110111
11111111   上から順にスキャン信号0から8までのキーの状態が示されている
11101111   0になっているところが，キーが押されていることを示している
11111101
11111111
11111110
> ⏎
> ⏎
> ⏎
> ⏎
> ⏎
>GD000 ⏎……………リターンキーを何回も押してから実行する
```

**図13.3 リスト13.3実行例**

### ▶ BIOS を使うとき

この章の最初でも述べたように，PSG や PPI はこれらを使うために BIOS が公開されています．ですからこれをコールすれば同じことができます．Challenge で使った I／O ポートを，BIOS ルーチンで扱うときは，次のルーチンを使います．

- ○ PSG のレジスタにデータを書き込む ……………WRTPSG(0093H)
- ○ スキャン信号を指定してキーの状態を調べる……SNSMAT(0141H)

# 14章

# 大量のデータ移動

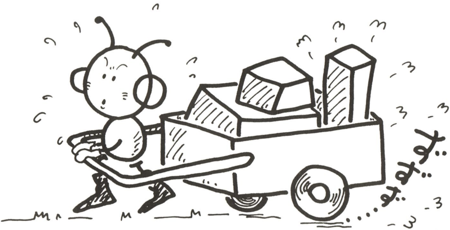
# 14.1

# ブロック転送とは

Z80のマシン語の命令の中には，BASICの命令では表現しにくいものがいくつかあります．12章で説明したビット操作命令というのもそのうちの1つですが，もう1つ**ブロック転送命令**といって，いっぺんに大量のメモリの内容を別のメモリのアドレスに移すというものがあります．

この命令はたった1つで，大きな仕事ができるというのが魅力です．しかしこの命令を使うには，前もっていくつかのレジスタに値(アドレスなどの情報)をセットしておかなければなりません．

いろいろめんどうなことはありますが，このブロック転送命令をマスターすると結構ハデなことができるので，プログラム作りがおもしろくなると思います．特にグラフィックスをマシン語プログラムで扱ってみたいと思っている人は，必ず使うようになるものですから，ぜひ覚えておきたいものです．



# 14.2

# ブロック転送命令の動作

ブロック転送には次にあげる2つがあります．どちらもオペランドはありません．

①ＬＤＩＲ
②ＬＤＤＲ

①のLDIR命令ではBC，DE，HLというAレジスタ以外の8ビットレジスタをすべて使います．それぞれ16ビットのデータを表しますが，その意味は次のとおりです．

ＨＬ……転送元のメモリのアドレス
ＤＥ……転送先のメモリのアドレス
ＢＣ……転送するメモリ内容の数(バイト)

これは"(HL)を(DE)にロードし，HLとDEを1増やすということを，BC回繰り返す"命令です．図14.1の(1)がこの命令を表すフローチャートで，(2)が実行した結果です．

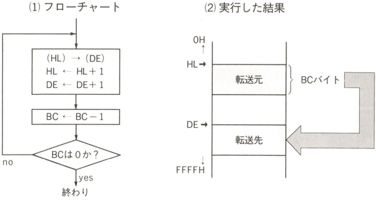



図14.1 LDIR命令のフローチャート

LDIR 命令は次のプログラムを実行したのと同じ働きをします．

## リスト 14.1

```
        PUSH    AF ............... Aレジスタの保存
L1:     LD      A,(HL)  } (HL) → (DE)
        LD      (DE),A  }
        INC     HL  } HLレジスタ,DEレジスタを1増やす
        INC     DE  }
        DEC     BC ..............BCレジスタを1減らす
        LD      A,B     } BCレジスタが0になったかどうかの確認
        OR      C       } "DEC  BC"はZフラグに影響を与えない
        JR      NZ,L1   }
        POP     AF
```

このようにふつうにプログラムを書くと 10 個も命令を並べなければならないのですが，LDIR 命令ではこれを 1 つの命令で処理することができます．

これはただプログラムが短くなるというばかりでなく，CPU が命令をメモリから読み出すのが 1 回ですむわけですから，**リスト 14.1** のようなプログラムを実行するよりもずっと効率がよくなります．ですからこのような場面ではなるべくブロック転送命令を利用した方がよいのです．

LDDR 命令も各レジスタの意味するものは同じです．しかし〝(HL)を(DE)にロードし，HL と DE を 1 減らすということを BC 回繰り返す″という点が違います．転送する数は BC レジスタに入れておきますが，**図 14.2** に示すように HL, DE レジスタにはそれぞれ転送元，転送先の領域のいちばん下のアドレスを指しておきます．

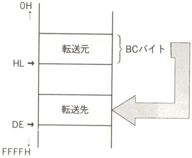



図14.2 LDDR命令実行のときの各レジスタの意味

## ブロック転送の応用

このブロック転送命令を使うときにはその動作をよく理解しておかなければなりません。特に，図14.3のように転送元の領域と転送先のメモリ領域が重なっているような場合には，LDIR命令とLDDR命令のどちらを使うかということに注意しなければなりません。

たとえばＤ８００Ｈ番地からＤＢＦＦＨ番地までのメモリの内容を，Ｄ９００Ｈ番地からＤＣＦＦＨ番地までに転送するとします。これを図14.3の(1)のようにHL, DEレジスタをセットしてLDIR命令を実行してはいけません。なぜなら，これでは最初にＤ８００Ｈ番地の内容をＤ９００Ｈ番地に移してしまうため，後でＤ９００Ｈ番地の内容をＤＡ００Ｈ番地に移そうとしたときに，前にあった値が失われているからです。

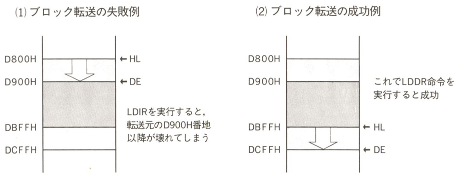



図14.3 領域が重なるときのブロック転送

こういった場合には，図14.3の(2)のようにＤＢＦＦＨ番地をHLレジスタ，ＤＣＦＦＨ番地をDEレジスタに入れて，LDDR命令を使わなければなりません。こうすると，転送元のメモリ領域の最下位アドレスから自分よりも後ろ(アドレスの大きい方)に内容を移すので，次の転送元を壊すということはなくなり，正しくデータの転送を行うことができます。

このように転送するメモリ領域が重なっているためにLDIR命令が使えないというのは，〝転送先の先頭アドレスが転送元の領域の中にある場合〟です．しかし，それ以外の場合には転送するメモリブロックの先頭をレジスタが指していた方がわかりやすいので，LDIR命令を使った方がよいでしょう．

ところでいまのLDIR命令の〝欠点〟を逆に利用して，大量のメモリに同じ内容のものを一度に書き込むテクニックがあります．図14.4を見てください．HLレジスタをあるアドレスに決め，DEレジスタはこの1つ下に置きます．そしてHLレジスタの指すメモリに00Hを入れておきます．ここでLDIR命令を実行するとどうなるでしょうか？

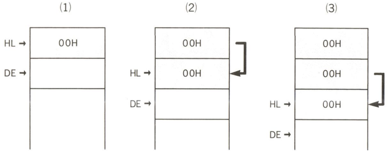



図14.4 LDIR命令によってメモリに同じ内容のものを一度に書き込む方法

まず最初に(HL)を(DE)に入れます(1)．次にHLとDEは1つずつ下にずれますが，このとき(HL)にはさっき上から移してきた00Hがはいっていますから，これを次のループのときに1つ下にまた移します(2)．以下この00Hを1つずつ下に移していきますから，結局全部でBC個のメモリをすべて00Hで埋めることができるのです(3)．

また同じように，最初にDEレジスタがHLレジスタの2つ先を指しているとどうなるでしょうか？ そうです．この場合は最初に(HL)と(HL＋1)にあった2バイトで，その後の領域が埋められます．このようにメモリをあるパターンで埋めるという場合には，ブロック転送命令が威力を発揮します．

## Challenge

### ▶ブロック転送命令の応用

ブロック転送を使ったプログラムを作る場合には，いろいろなポイントがありますが，ここでは実際にそれを試してみることにします．

まず最初にブロック転送命令によって転送されるメモリ領域というものがなければなりません．これは大量にあった方が，いかにもブロック転送をやったという気持になれるのですが，あまりにたくさんのメモリ領域をモニタのSコマンドで入力していくのはたいへんです．そこで最初にこの転送される領域を作るプログラムを作ります．

**テーマ** D800Hからの256バイトをD900Hからに移す

これはブロック転送の非常に簡単な例です．まず転送元の領域と転送先の領域がまったく分かれていますから，ふつうにブロック転送をすればよいのです．リスト14.2がプログラムで，これを実行したものが図14.5です．

### リスト 14.2

```
D000 2100D8    SETMEM: LD    HL,0D800H
D003 3E00              LD    A,0
D005 77        SETME1: LD    (HL),A     ┐ループ内でメモリに 0H～FFH
D006 23                INC   HL         │を書き込む
D007 3C                INC   A          ┘
D008 20FB              JR    NZ,SETME1 ……Aレジスタが FFH から
                                          INC命令で0になったら
                                          次の命令に進む
D00A 2100D8            LD    HL,0D800H ……転送元の先頭アドレス
D00D 1100D9            LD    DE,0D900H ……転送先の先頭アドレス
D010 010001            LD    BC,100H   ……転送バイト数
D013 EDB0              LDIR
D015 C9                RET
```

```
>SD000 ⏎
D000 FF-21 FF-00 FF-D8 FF-3E
D004 FF-00 FF-77 FF-23 FF-3C
D008 FF-20 FF-FB FF-21 FF-00      プログラムの入力
D00C FF-D8 FF-11 FF-00 FF-D9
D010 FF-01 FF-00 FF-01 FF-ED
D014 FF-B0 FF-C9 FF-.
>DD000 D01F ⏎
D000 21 00 D8 3E 00 77 23 3C   !.リ>.w#<
D008 20 FB 21 00 D8 11 00 D9    ミ!.リ..ル   プログラムの確認
D010 01 00 01 ED B0 C9 FF FF   ...\ー/■■
D018 FF FF FF FF FF FF FF FF   ■■■■■■■■

>GD000 ⏎……………実行

>DD800 ⏎……………結果を確認する
D800 00 01 02 03 04 05 06 07   ........
D808 08 09 0A 0B 0C 0D 0E 0F   ........
D810 10 11 12 13 14 15 16 17   ........
D818 18 19 1A 1B 1C 1D 1E 1F   ........   転送元のデータ(一部)
D820 20 21 22 23 24 25 26 27    !"#$%&'
D828 28 29 2A 2B 2C 2D 2E 2F   ()*+,-./
D830 30 31 32 33 34 35 36 37   01234567
D838 38 39 3A 3B 3C 3D 3E 3F   89:;<=>?

>DD900 ⏎
D900 00 01 02 03 04 05 06 07   ........
D908 08 09 0A 0B 0C 0D 0E 0F   ........
D910 10 11 12 13 14 15 16 17   ........
D918 18 19 1A 1B 1C 1D 1E 1F   ........   転送先に移されたデータ(一部)
D920 20 21 22 23 24 25 26 27    !"#$%&'
D928 28 29 2A 2B 2C 2D 2E 2F   ()*+,-./
D930 30 31 32 33 34 35 36 37   01234567
D938 38 39 3A 3B 3C 3D 3E 3F   89:;<=>?

>■
```

**図14.5 リスト14.2実行例**

どうです，うまくいきましたか？ この場合は LDIR 命令を使っても，LDDR 命令を使っても結果は同じです．ただし LDDR 命令を使うときは，HL レジスタと DE レジスタにはそれぞれD８ＦＦＨとD９ＦＦＨをロードしておきます．

# 15章

# BASICとマシン語の結合

# 15.1
# マシン語プログラムとBASIC

これまで，マシン語の命令を見ながらChallengeでいくつかプログラムを作ってみましたが，これらはみなモニタから入力，実行していました．しかし，いつまでもこのような方法をとっていたのでは不便です．なぜなら，モニタがあればマシン語のプログラムを実行できるということは，逆にいえばこれまでに作ったプログラムはモニタがないと実行することができないということだからです．

これからマシン語の腕が上達すると，グラフィック画面を使ったり，大きなプログラムを書いたりできるようになります．そんな場合に，いちいちこのモニタをロードしてからでないと，そのマシン語プログラムが動かせないというのでは不便です．

またプログラムを作るのに，BASICとマシン語の両方を使いたいということもあります．前にも述べましたが，BASICにはBASICの，マシン語にはマシン語のよさがあります．大きなプログラムを作るときには，BASICでできるようなことはBASICで，スピードが要求される部分はマシン語でと，それぞれの特徴を活かして分担してプログラムを作るということがよくあります．

こういう場合，BASICプログラムも作らなければならないのですが，モニタプログラム(これはBASICで書かれている)があると，それができなくなります．このように，モニタをロードするのがわずらわしいどころか，〝あっては困る〞ということもあるのです．

そこでこの章ではモニタから離れて，マシン語を使う方法を見ていきます．ここでのポイントは次の2つです．

① マシン語プログラムをどうやってメモリに入れるか
② マシン語プログラムをどう実行させるか

# 15.2

# プログラムの作成

1つのプログラムをBASICとマシン語を使って作る場合には，BASICプログラムがメインルーチンになり，そこからマシン語のプログラムをサブルーチンコールするようにします．

**図15.1**のように，BASICは複数のマシン語のプログラムを呼び出すことができます．マシン語はその中でまたマシン語をサブルーチンコールするということはできますが，マシン語の方からBASICプログラムをコールすることはできません．

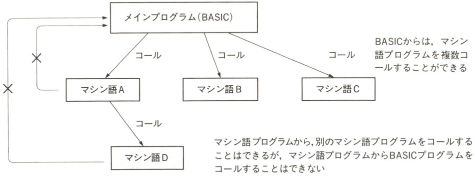



**図15.1 BASICとマシン語の関係**

## プログラムの作り方

プログラムを作る上では，ふつう設計→コーディング→デバッグというようにいくつかのステップを踏みますが，マシン語を使う場合も同様です．特に今回のように，BASICのサブルーチンとして働くマシン語プログラムを作ろうというのなら，マシン語とBASICを別々に開発する必要があります．

# 15.3

# マシン語をメモリに入れる

マシン語のプログラムは，メモリに格納されていないと実行できません．しかもそのアドレスはあらかじめプログラムを作る段階で決めておくので，正確にそのアドレスにはいっていなければなりません．

モニタを使う場合には，マシン語プログラムをSコマンドで1バイトずつセットしていました．しかしモニタがない場合には別の方法を考えなければなりません．

MSX-BASICでは，直接メモリに値（データやマシン語プログラム）を書き込むには次の2つの方法があります．

① POKE文を使う
② テープにセーブしたデータやプログラムをBLOAD命令でロードする

ふつうは①の方法をとるのが一般的です．POKE文は2章でも見ましたが

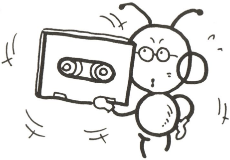
```
POKE &HD000,&H89
```

というようにメモリのアドレスとそこに入れたい値を指定します．

②のBLOAD命令はBASICプログラムをロードするときのように

```
BLOAD "ファイルスペック*"
```

というように使ってテープやディスクからファイルを読み込み，直接メモリに書き込みます．しかしこの命令に読ませるプログラムファイルは，あらかじめ連続したメモリの内容をセーブするBSAVE命令で作っておかなければなりません．



＊デバイス名＋ファイル名のこと．カセットの場合デバイス名は"CAS:"となる．

## POKE 文を使う

POKE 文は1回の実行で1バイトのデータしか書き込むことができません．ですからマシン語のプログラムを作った(この〝作った〟というのは，ハンドアセンブルをしてマシンコードにしたという意味)ときに，いちいちこの POKE 文を使っていたのではたいへんです．そこで，ふつうは**リスト 15.1** のように FOR〜NEXT ループを使ったプログラムを作ります．

**リスト 15.1**

```
3000 REM MACHINE CODE          ……マシンコードの先頭アドレス
3010 FOR I=&HD000 TO &HD035    ……マシンコードの終了アドレス
3020  READ D$                  } 16進数をデータから読み込んで数値に変換する
3030  D=VAL("&H"+D$)           }
3040  POKE I,D
3050 NEXT
3060 RETURN
3070 DATA 21,00,D0,01,40,D8,11,34  } DATA文のデータにマシンコードを
3080 DATA DD,ED,B0,44 ……           } 16進数で書いておく
 :
```

マシン語のプログラムはメモリに連続してはいりますから，**リスト 15.1** のようにループを使ってメモリに書き込みます．ハンドアセンブルしたプログラムを DATA 文のデータとして 16 進数で書いておき，それを読み出して数値に変換し，POKE 文で書き込みます．

## マシン語プログラムのファイルを作る

**リスト 15.1** を実行すると，メモリにプログラムがはいりますから，今度はそれをテープやディスクにセーブします．BSAVE 命令は次のように3つの引数をとります．

BSAVE "CAS：PROG1",&HD000,&HD028〔,&HD000〕

ファイル名の後の2つの引数はメモリの何番地から何番地をセーブするかを指定し

ます．その次のものは〝実行開始アドレス〟といって，ロードしたときにオートスタートするプログラムを作りたいときに指定します．この本ではオートスタートさせるようなプログラムは作りませんから，この最後の引数は考えなくてもよいでしょう．

こうしてテープかディスクにマシン語プログラムのファイルを作ったら，今後は

BLOAD 〝CAS：PROG1〟……カセットテープからのロード

でそのプログラムをロードできます．

## プログラム全体のファイルの作り方

BASIC＋マシン語のプログラムを作る場合，上でやったようにマシン語プログラムのファイルを作ると，BASIC とマシン語で別々のファイルを作ることになります．つまり，

BLOAD 〝CAS：PROG1〟……マシン語プログラムのロード
LOAD 〝CAS：PROG2〟……BASIC プログラムのロード

というように2つの LOAD 命令を実行しなければロードできません．

これがめんどうな場合には，マシン語のプログラムを DATA 文として，BASIC プログラムの一部にする方法があります(リスト 15.2)．

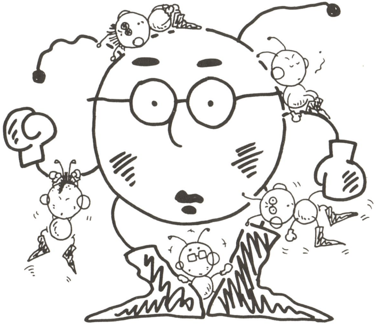
### リスト 15.2

```
1000 CLEAR 300,&HCFFF
1010 GOSUB 3000 ……………最初にマシンコード書き込みサブルーチンを呼び出す
1020 REM MAIN START ……………ここからメインルーチンが始まる
  ⋮
3000 REM MACHINE CODE          ┐
3010 FOR I=&HD000 TO &HD035    │
3020  READ D$                  │ マシンコード書き込みサブルーチン
3030  D=VAL("&H"+D$)           │
3040  POKE I,D                 │
3050 NEXT                      ┘
3060 RETURN
3070 DATA 21,00,D0,01,40,D8,11,34
3080 DATA DD,ED,B0,44
  ⋮
```

〔このプログラムは動きません〕

この方法をとれば，マシン語のプログラムをBASICのDATA文に入れてしまうので，テープからのロードは1回ですみます．その代わり，BASICのプログラム全体がこのDATA文のために大きくなってしまいますし，BASICプログラムを実行するときにメモリに書き込みを行うので，よけいな時間がかかります．

マシン語のプログラムが大きくなる場合には別々のファイルにセーブし，短い場合にはDATA文に入れて，BASICプログラムの最初にPOKE文でメモリに入れるというようにすればよいと思います．



# 15.4

# マシン語プログラムのコール

マシン語プログラムをメモリに入れたら，今度はそれをBASICプログラムからサブルーチンコールすることを考えてみましょう．これにはUSR関数というものを使いますが(モニタでも使っている)，この関数を使う場合には，いくつか重要な注意があります．

## マシン語プログラムの保護

まず最初に，メモリに入れたマシン語のプログラムを保護しなければなりません．メモリの中にはいま入れたマシン語プログラムのほかに，BASICの変数領域*などが取られます．これはふだんBASICプログラムだけを使っているときは気になりませんが，マシン語を使うときには気を付ける必要があります．

マシン語プログラムを入れたアドレスに変数領域が重なってしまうと，せっかくのプログラムが壊れてしまいます．そこでこのマシン語プログラムを変数領域によって壊されることを防ぐために，CLEAR文を使います．

CLEAR文は変数の初期化を行う命令ですが，そのほかに変数領域を〝あるアドレスよりも上(アドレスの小さい方)に置け〟ということを指定します．図15.2にあるように，BASICのプログラムはアドレスの若い方からはいります．そこでマシン語のプログラムというのはなるべく下の方(アドレスの大きい方)に置くことにします．ただしF380H番地から下はMSX-BASICのワークエリア**として使われるので使ってはいけません．

---

*　BASICプログラムが実行されるとき，その中で使われる変数の値をとっておくためのメモリ
** BASICインタープリタが動くために使うデータを入れておく領域

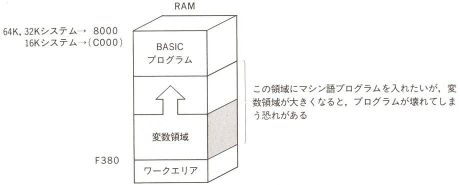



図15.2 メモリマップ

きっちりとマシン語プログラムの下をＦ３７ＦＨに付けるという必要はありません．これまでやってきたように，Ｄ０００Ｈ番地から始まるくらいで十分でしょう．ただし，こうしたら，BASICの中で使う変数領域はこれよりも上にできなければ困ります．そこで

```
CLEAR 300,&HCFFF
```

という命令をBASICプログラムのいちばん最初に入れます．こうするとメモリは図15.3のような状態になり，変数領域はＣＦＦＦＨ番地よりも上に取られますから，マシン語プログラムが壊されることはありません．



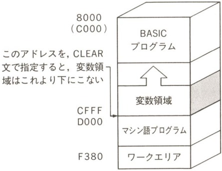



図15.3 メモリマップ(CLEAR文実行後)

## マシン語プログラムの呼び出し

さてマシン語プログラムを保護できたら，次はそのプログラムを BASIC からコールする方法を見てみましょう．BASIC からマシン語プログラムをコールするためには，マシン語の開始アドレスを教えてやらなければなりません．これには DEF USR という宣言文を使います．これは次のようにして，マシン語プログラムの開始アドレスを宣言します．

```
DEF USR=&HD000
```

これでマシン語のプログラムは，D000H番地から始まることが宣言できました．この宣言をしたらその後，

```
X=USR(0)
```

という代入文を実行すると，そのマシン語プログラムを宣言したアドレスから呼び出します．つまりマシン語プログラムが実行されるのです．これは 6 章でも述べたようにサブルーチンコールですから，マシン語の RET 命令に出会うと，また BASIC のこの代入文のところに戻ってきます．

USR 関数は 0 から 9 まで 10 種類宣言できますが，それぞれを区別するために USR1 とか USR5 というように後ろに番号を付けます．この番号を付けないと，0 が付けられたという意味になります．

ここでいくつか疑問が生じます．まず何のために代入文にするのか，またカッコの中の 0 という値は何なのかということです．

USR 関数では，単に BASIC からマシン語プログラムをコールするというだけでなく，BASIC とマシン語のプログラムの間でお互いにデータを渡し合うことができるようになっています．代入文になっているのはマシン語から BASIC の変数に値を渡す(代入する)ためであり，引数(カッコの中の値)は逆に BASIC からマシン語に値を渡すためのものなのです．

最初のマシン語から BASIC へ渡すというのは難しいのでこの本では使いませんが，BASIC からマシン語に引数を渡すという方法は，この次の節で説明します．引数

を使わずにただマシン語プログラムを実行するということももちろんできますが，この場合引数を付けないとエラーが起こってしまうので，適当な値(ダミー)をカッコの中に書いておきます．

## Challenge

それではここで実際にBASICとマシン語のプログラムを結合したものを作ってみましょう．ここで使うマシン語のプログラムは，文字で画面に四角を描くというものを使います．

**リスト15.3**がマシン語のプログラムです．これをBASICプログラムからコールします．まず最初にマシン語プログラムをメモリに書き込む作業が必要ですが，短いのでBASICのDATA文に入れてPOKE文で書き込みます．**リスト15.4**の3000行からのサブルーチンがそうですが，いままではモニタのSコマンドで入力していたマシンコードをDATA文に入れておけばよいのです．

BASICのプログラムでは画面の初期設定などをしてから，このマシン語プログラム(D000H番地から)をサブルーチンコールしています．マシン語では四角を描くとすぐにBASICに戻りますが，そこですぐにプログラムが終わってしまうと，画面に"Ok"という文字が表示されてきたなくなるので，1060行目に無限ループを入れておきます．この部分はBASICですから，ストップをかければプログラムは止まります．

**リスト 15.3**

```
D000 0608     SQU:    LD     B,8    ……………X座標(外側のループ)
D002 C5       SQU1:   PUSH   BC
D003 60               LD     H,B
D004 24               INC    H
D005 0607             LD     B,7    ……………Y座標(内側のループ)
D007 68       SQU2:   LD     L,B
D008 2C               INC    L
D009 CDC600           CALL   POSIT  ………カーソルの位置指定
D00C 3E41             LD     A,41H  ………"A"という文字を表示する
D00E CDA200           CALL   CHPUT
D011 10F4             DJNZ   SQU2
D013 C1               POP    BC
D014 10EC             DJNZ   SQU1
D016 C9               RET
```

## リスト 15.4

```
1000 CLEAR 300,&HCFFF
1010 GOSUB 3000
1020 REM MAIN
1030 CLS
1040 DEFUSR=&HD000   } マシン語プログラムをコールする
1050 X=USR(0)        }
1060 GOTO 1060
3000 REM MACHINE CODE
3010 FOR I=&HD000 TO &HD016   }
3020  READ D$                 }
3030  D=VAL("&H"+D$)          } マシンコードの書き込み
3040  POKE I,D                }
3050 NEXT                     }
3060 RETURN
3070 DATA 06,08,C5,60   }
3080 DATA 24,06,07,68   }
3090 DATA 2C,CD,C6,00   } マシンコード
3100 DATA 3E,41,CD,A2   }
3110 DATA 00,10,F4,C1   }
3120 DATA 10,EC,C9      }
```

```
AAAAAAAA
AAAAAAAA
AAAAAAAA
AAAAAAAA
AAAAAAAA
AAAAAAAA
AAAAAAAA









color   auto   goto   list   run
```

図15.4 リスト15.4実行例

# 15.5

# 引数を渡す

USR 関数には，カッコの中の引数を使って BASIC からマシン語のプログラムに値を渡すという機能があります．引数というのは，同じサブルーチンをコールする場合に，渡す値によっていろいろ処理方法を変えて欲しいときに使うものです．

USR 関数では BASIC からマシン語に，1つの引数を渡すことができますが，これを使うには，USR 関数によってマシン語のプログラムが呼び出された時点で，各レジスタがどういう状態になっているかということを理解しなければなりません．

## USR 関数の引数

USR 関数に引数を付けてマシン語をコールすると，**図 15.5** に示すように，HL レジスタがあるメモリのアドレスを指し，その中に引数として渡した数がはいります．

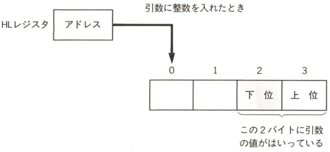



**図15.5 USR関数をコールしたときの状態**

BASIC で扱うデータには整数のほかに実数，倍精度実数，それに文字などがありますが，**図 15.5** は整数の場合です．実数はメモリの中で非常に複雑な構造をしているので，マシン語でこれを扱うのはたいへんです．そこでここでは整数だけを扱うことに

しますが，引数が整数になるようにするには，BASICのプログラムの方でそれを指定(宣言)しておかなければなりません．

図15.6のようにUSR関数の引数が整数の場合には，HLレジスタの指すメモリの2バイト先にその整数の下位8ビット，その次のアドレスに上位8ビットがはいっています．ですからマシン語のプログラムの最初にこのメモリの値を取り出せば，BASICプログラムからの引数を受け取ることができます．

## Challenge

ここでは前の節で作ったプログラムを改良して，次のようなプログラムを作ってみます．

**テーマ** キャラクタコードを入力して，その文字で四角を描く

キャラクタコードを入力する部分はBASICで，四角を描く部分はマシン語で作ります．リスト15.5とリスト15.6がそのプログラムです．

### リスト15.5

```
D000 23       SQU:   INC    HL       } BASICからの引数をCレジスタ
D001 23              INC    HL       } に入れておく
D002 4E              LD     C,(HL)   }
D003 0608            LD     B,8
D005 C5       SQU1:  PUSH   BC
D006 60              LD     H,B
D007 24              INC    H
D008 0607            LD     B,7
D00A 68       SQU2:  LD     L,B
D00B 2C              INC    L
D00C CDC600          CALL   POSIT
D00F 79              LD     A,C  ........ そのキャラクタコードで
D010 CDA200          CALL   CHPUT         四角を描く
D013 10F5            DJNZ   SQU2
D015 C1              POP    BC
D016 10ED            DJNZ   SQU1
D018 C9              RET
```

## リスト 15.6

```
1000 CLEAR 300,&HCFFF
1010 DEFINT X ……………必ずXを整数型に宣言しておく
1020 GOSUB 3000
1030 REM MAIN
1040 CLS
1050 INPUT "CHARACTER CODE";X ……………キャラクタコードを入力する
1060 DEFUSR=&HD000
1070 X=USR(X) ……………カッコの中のXの値が引数としてマシン語ルーチンに渡される
1080 GOTO 1080
3000 REM MACHINE CODE
3010 FOR I=&HD000 TO &HD018
3020   READ D$
3030   D=VAL("&H"+D$)              マシンコードの書き込み
3040   POKE I,D
3050 NEXT
3060 RETURN
3070 DATA 23,23,4E,06
3080 DATA 08,C5,60,24
3090 DATA 06,07,68,2C
3100 DATA CD,C6,00,79        マシンコード
3110 DATA CD,A2,00,10
3120 DATA F5,C1,10,ED
3130 DATA C9
```

マシン語の方では，まずCレジスタにHLレジスタの指すメモリの2バイト先の内容を読み出します．ここにはBASICで入力されたキャラクタコードがUSR関数によって入れられるはずですから，このキャラクタコードの文字を使って四角を描きます．

引数は2バイトですが，キャラクタコードは8ビットなので上位バイトは必ず0になりますし，この場合は読み出す必要がありません．

BASICの方ではINPUT文によってまずキャラクタコードを入力して，それを引数にしてマシン語のプログラムをコールします．

このプログラムを実行したのが図15.6です．最初にキャラクタコードを尋ねてきますから，そこで32から255までの適当な数字を入れてやります．すると画面にその数字のキャラクタコードの文字で四角が描かれます．

マシン語プログラム中ではBIOSのCHPUTルーチンを使っていますから，0から31までのコントロールコードを引数にすると，四角形が描かれずに，四角形の大きさの回数だけコントロールコードに相当する仕事を行います．

また 256 以上の数字を入れると，その数を 256 で割った余りのコードを入力したのと同じことになります．これはマシン語の方で下位 8 ビットしか見ていないからです．

```
CHARACTER CODE? 63
 ????????
 ????????
 ????????
 ????????
 ????????
 ????????
 ????????

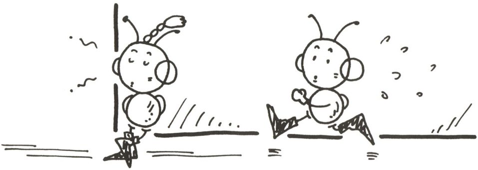



color   auto   goto   list   run
```

キャラクタコードを入力する

そのキャラクタで四角を描く
"?"のキャラクタコードは63

図 15.6 リスト 15.6 実行例



# 16章

# マシン語プログラムの応用

# 16.1

# 乱数を作る

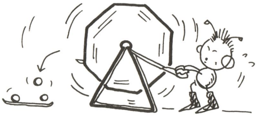
この節では乱数の作り方を紹介します．乱数というのは，プログラムを作る人，プログラムを使う人が〝予知できない数〟のことです．BASIC には 0 から 1 の間の実数の乱数を発生させるものとして，RND 関数というものがあります．これは，

```
C=INT(RND(1)*10)+1
```

というように書くと，1 から 10 までのデタラメな整数を作ることができます．

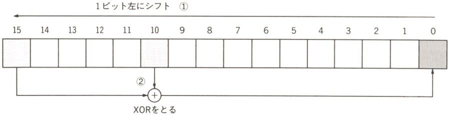
乱数の応用はいろいろ考えられますが，マシン語で乱数が欲しくなるというのは，多くの場合ゲームを作っているときでしょう．ゲームプログラムは，プレーヤー側の動きはキーボードからの入力によって決まります．しかし敵(つまりコンピュータ側のプレーヤーや宇宙船)の動きというのも，何らかの方法で決めなければなりませんから，毎回相手が同じ動きをするというのでは，ゲームとしてのおもしろさが失われてしまいます．そこで〝毎回予測できない動き〟を得るために，乱数というものが必要になるのです．

マシン語で乱数を作る方法はいくつかありますが，ここではそのうちの 1 つ，〝M 系列〟というものを紹介します．これは 2 バイトの整数の乱数を発生させるプログラムです．RND 関数では実数で出ますが，マシン語ではむしろ実数よりも整数の方が扱いやすいので，整数の乱数が発生できた方が便利だと思います．



## M 系列の原理

“M 系列”というのは，乱数を発生させるための 1 つの考え方です．これは図 16.1 のように，あるビット列があるときにこの中で乱数を作ります．




ここでは 2 バイトの整数の乱数を発生させるために 16ビットのビット列を用いているが，実際には何ビットでもかまわない．

図16.1 M系列

M 系列の乱数を使用するときは，まず乱数を発生させるもととなるデータを覚えておく場所を確保します(ふつうはメモリの中に 2 バイト取っておく)．そして乱数を発生するサブルーチンをコールすると，そのたびにその 16 ビットから新しい乱数を作り出し，その値をリターンパラメータとしてメインルーチンに返し，同時にメモリにもその値を取っておき，次の乱数に備えます．

乱数発生サブルーチンでは，次のような操作が行われます．

**(1) 全ビットを左にシフトする**

**(2) シフトする前のビット 15 とビット X の XOR をとって，それをビット 0 に入れる**

(2)の X というのは，0 から 14 までの任意のビットです．つまり，ビット 15 以外のビットを取りさえすれば，どれを選んでも乱数を得ることができます．

乱数を使うためには，事前に乱数の初期値として 0 以外の任意の 16 ビットの値をそのメモリに書き込んでおきます．そうするとその後は，上の操作を行う乱数発生サブルーチンをコールするたびに，16 ビットの乱数を発生します．

M 系列の原理では，上の方法によって 0 以外のすべての 16 ビットのパターンが発生することになっています．0 に対して上の操作を行うと，その後 0 しか出なくなってしまいますので，必ず最初は 0 以外の数を入れておかなくてはなりません．

どうしてこういう操作をすると 16 ビットの乱数が得られるのでしょうか？ これが M 系列の不思議なところです．実際には証明されているのですが，ここではそれを紹介しません．もし興味があれば，数学の本でも探してみてください．

ここではとにかく〝こうすれば乱数ができる〟ということが大切ですから，上の原理を応用したプログラムを作ってみましょう．

## プログラム RANDOM

M 系列の原理に基づいて，マシン語のプログラムを作ったのが，**リスト 16.1** です．これはサブルーチンの形になっていて，このサブルーチン(RANDOM)をコールするたびに 16 ビットの乱数が HL レジスタにはいって戻ってきます．

### リスト 16.1

```
D000 F5        RANDOM: PUSH  AF
D001 C5                PUSH  BC
D002 AF                XOR   A         } A,Cレジスタをともに0にしておく
D003 4F                LD    C,A       }
D004 2A18D0            LD    HL,(RND)  …このプログラムの下(D018H番地)の2バイトに乱数を入れておく
D007 29                ADD   HL,HL     ………左に1ビットシフトする
D008 17                RLA             ………………ビット15をAレジスタに入れておく
D009 CB54              BIT   2,H
D00B 2801              JR    Z,RAN1
D00D 0C                INC   C         ………ビット10はCレジスタに入れる
D00E A9        RAN1:   XOR   C         ………2つのビットのXORをとる
D00F 2801              JR    Z,RAN2
D011 23                INC   HL
D012 2218D0    RAN2:   LD    (RND),HL  …次回のためにとっておく
D015 C1                POP   BC
D016 F1                POP   AF
D017 C9                RET
      ↓―――――― RNDの指すアドレス
D018 0100

RNDは，その下のD018H番地からとればよいことがわかる．
これはラベルと同じ扱いでできる
```

メモリのＤ０１８Ｈ番地が，２バイトの乱数を貯えておくところで，毎回この値から乱数を作り出します．このプログラムではビット 15 とビット 10 を取り出して，その XOR をビット 0 に入れています．プログラムの中の〝BIT　2，H〟を書き換えれば，別のビットを取り出すことができます．

## プログラムのテスト

このサブルーチンではどんな乱数ができるのかを，実際に見てみましょう．とりあえずモニタで実行してみます．これは HL レジスタの内容を表示してみるためです．Ｄ８００Ｈ番地から，**リスト 16.2** のように乱数ルーチンをコールして，HL レジスタの内容を表示するプログラムを作ります．

**リスト 16.2**

```
D800 CD00D0            CALL    RANDOM  …サブルーチン"RANDOM"をコールする
D803 CD08E0            CALL    PUTHL   …サブルーチン"PUTHL"をコールする
D806 C9                RET
```

これを入力したら，**図 16.2** に示すように実行する前に乱数の初期値を入れておきます．ここではＤ０１８Ｈ番地に０１Ｈ，Ｄ０１９Ｈ番地に００Ｈを入れます（これは 16 ビットで〝1〟を意味する）．そして〝ＧＤ８００Ｈ〟によって，プログラムを実行すると，**図 16.2** のように，実行するたびに 2，4，8…と乱数が表示されます．

このようにサブルーチン RANDOM は，別のルーチンからコールされるたびに乱数を発生します．発生される乱数は 0 以外の 16 ビットの数すべて，つまり 65535 種類の数が出ます．

この M 系列の方法では，乱数の初期値に何を入れるかによって，その後の乱数の出方が決まります．このような乱数を擬似乱数といいます．擬似乱数は，初期値によって以後の乱数が予測されるという欠点がありますが，偏りのない一様な乱数が得られます．

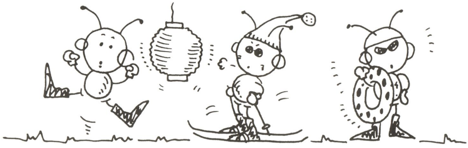
このルーチンの利用法は，メインルーチンで乱数が必要になったところでこのサブルーチンをコールするだけです．そうすると HL レジスタに 16 ビットの乱数が得られ，そのほかのレジスタはすべて保存されます．

```
>SD000 ⏎
D000 FF-F5 FF-C5 FF-AF FF-4F
D004 FF-2A FF-18 FF-D0 FF-29
D008 FF-17 FF-CB FF-54 FF-28      リスト16.1のプログラム
D00C FF-01 FF-0C FF-A9 FF-28
D010 FF-01 FF-23 FF-22 FF-18
D014 FF-D0 FF-C1 FF-F1 FF-C9
D018 FF-01 FF-00 FF-.
>DD000 D017 ⏎                     ←乱数の初期値
D000 F5 C5 AF 4F 2A 18 D0 29  ΦナッO*.ミ)
D008 17 CB 54 28 01 0C A9 28  .ヒT(..ゥ(     乱数プログラムの確認
D010 01 23 22 18 D0 C1 F1 C9  .#".ミチむノ

>SD018 ⏎
D018 01-__ 00-__ FF-.             乱数の初期値は0001Hであることを確認する
>SD800 ⏎
D800 00-CD 00-00 00-D0 00-CD      乱数プログラムをコールして出力するプログラム
D804 00-08 00-E0 00-C9 00-.       (リスト16.2)
>DD800 D807 ⏎
D800 CD 00 D0 CD 08 E0 C9 00  ヘ.ミヘ.たノ. ……そのプログラムを確認する

>GD800 ⏎
0002
>GD800 ⏎
0004
>GD800 ⏎
0008                              実行するたびに乱数が出る
>GD800 ⏎
0010
>GD800 ⏎
0020
>■
```

図16.2 リスト16.1, 16.2実行例



# 16.2
# PSGを使って音を出す

13章のI/Oポートのところでも，PSGにデータを送って音を出してみました．あのときはI/Oポートを直接操作して，PSGのレジスタとデータをポートへ出力していました．しかしBIOSの中に，PSGのレジスタに書き込みを行うものがありますから，ここではそれを使ってみましょう．

## PSGを使う方法

PSGに関するBIOSルーチンには表16.1のようなものがあります．

| アドレス | 名　前 | 機　　能 |
|---|---|---|
| 0093 | WRTPSG | PSGレジスタに値を出力する<br>●エントリーパラメータ：Aレジスタに PSGレジスタの番号を入れる．Eレジスタに出力する値を入れる<br>●リターンパラメータ　：なし<br>●使用レジスタ　　　　：なし |
| 0096 | RDPSG | PSGレジスタの値を読む<br>●エントリーパラメータ：Aレジスタに PSGレジスタの番号を入れる<br>●リターンパラメータ　：Aレジスタに PSGレジスタから入力した値がはいる<br>●使用レジスタ　　　　：なし |

表16.1 PSGのBIOSエントリー

2番目のWRTPSGがBASICのSOUND文にあたるもので，PSGのレジスタへの書き込みを行います．RDPSGというのは，PSGのレジスタの値を読み出すものです．PSGに関するBIOSルーチンはこれだけしかありません．

## SOUND プログラム

ここでは〝キーボード鍵盤〟を作ってみましょう．これはキーボードをピアノの鍵盤に見立てて，キーを押せばそれに相当する音が出るというプログラムです．プログラム自体は簡単で，どのキーが押されているかを調べて，それによって音を出すということを繰り返せばよいのです．

ここで問題になるのは，ド，レ，ミ…という音を出すにはどのような値をレジスタに書き込んでやればよいかということです．ここでは1オクターブの音が出ればよいでしょうから，**表 16.2** にある音が必要になります．

| 音　階 | ド | レ | ミ | ファ | ソ | ラ | シ | ド |
|---|---|---|---|---|---|---|---|---|
| PSGの周波数 | 03BAH | 0352H | 02F5H | 02CBH | 027DH | 0237H | 01F9H | 01DDH |

**表16.2　1オクターブの音**

PSG では 2 つのレジスタを使って，2 バイト分の周波数の音を出すことができます．たとえば〝ド〟の音を出そうと思ったら，PSG のチャンネル A(レジスタ 0 と 1)に０３H，ＢＡHという値を書き込んで，レジスタ 7 をＦＥHにしてその音を出すようにすればよいのです．このプログラムで使うキーは**図 16.3** のとおりで，それぞれ図に書き込んである音が出るようにします．

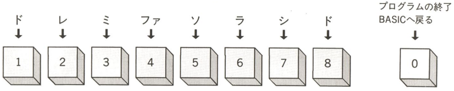



**図16.3　キーボードと音の対応**

このプログラムは**リスト 16.3** のマシン語プログラムを**リスト 16.4** の BASIC の DATA 文として組み込んでいます．BASIC で行うことはマシン語をメモリにセットして画面にキーボードの説明を書くだけで，実際に押されているキーを調べて音を出す部分はマシン語で行います．

## リスト 16.3

```
D000 3E08             LD     A,8            ポリューム指定
D002 1E0A             LD     E,10
D004 CD9300           CALL   WRTPSG
D007 3E07             LD     A,7            最初は音を消しておく
D009 1EFF             LD     E,0FFH
D00B CD9300           CALL   WRTPSG
D00E 3E00    PSGLP:   LD     A,0            [0]～[7]のキーを調べる
D010 CD4101           CALL   SNSMAT
D013 CB47             BIT    0,A            [0]が押されていたらBASIC
D015 C8               RET    Z              へ戻る
D016 E6FE             AND    0FEH           ビット7～ビット1のどれかが
D018 EEFE             XOR    0FEH           0のものであったらPSG1へジ
D01A 2016             JR     NZ,PSG1        ャンプする([1]から[7]のキー
                                            が押されている)
D01C 3E01             LD     A,1            [8]のキーが押されていたら
D01E CD4101           CALL   SNSMAT         PSG0へジャンプする
D021 E601             AND    01H
D023 2805             JR     Z,PSG0
D025 CD4ED0           CALL   STOP           音を消してループに戻る
D028 18E4             JR     PSGLP
D02A 1101DD  PSG0:    LD     DE,0DD01H      ド(1オクターブ上の方)の
D02D CD56D0           CALL   SOUND          データで音を出す
D030 18DC             JR     PSGLP
D032 EEFE    PSG1:    XOR    0FEH
D034 0607             LD     B,7
D036 0E02             LD     C,2 ……………………調べるビット
D038 2169D0           LD     HL,0D069H ……サウンドのデータのはいっ
                                            ているアドレス
D03B 5F      PSG2:    LD     E,A
D03C A1               AND    C              調べたビットが0なら(キーが押さ
D03D 7B               LD     A,E            れていたら)PSG3へジャンプす
D03E 2806             JR     Z,PSG3         る
D040 23               INC    HL             アドレスを2つ進める
D041 23               INC    HL
D042 CB01             RLC    C ……………調べるビットを1つずらす
D044 10F5             DJNZ   PSG2
D046 5E      PSG3:    LD     E,(HL)         D069H番地からにあるサウンド
D047 23               INC    HL             データ(2バイト)がDEレジスタ
D048 56               LD     D,(HL)         にはいる
D049 CD56D0           CALL   SOUND
D04C 18C0             JR     PSGLP
D04E 3E07    STOP:    LD     A,7 …………音を消す
D050 1EFF             LD     E,0FFH
D052 CD9300           CALL   WRTPSG
D055 C9               RET
D056 3E01    SOUND:   LD     A,1
D058 CD9300           CALL   WRTPSG
D05B 3E00             LD     A,0
```

次頁へ続く

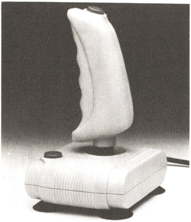
```
D05D 5A          LD     E,D         ┐ DEレジスタの内容をそれぞれ
D05E CD9300      CALL   WRTPSG      │ PSGのレジスタ0とレジスタ1
D061 3E07        LD     A,7         │ に書き込み,音を出す
D063 1EFE        LD     E,0FEH      │
D065 CD9300      CALL   WRTPSG      ┘
D068 C9          RET
D069 03BA               03H,0BAH ……ド  ┐
D06B 0352               03H,52H  ……レ  │
D06D 02F5               02H,0F5H ……ミ  │
D06F 02CB               02H,0CBH ……ファ│ D069H番地からD076H番地
D071 027D               02H,07DH ……ソ  │ にサウンドデータを入れる
D073 0237               02H,037H ……ラ  │
D075 01F9               01H,0F9H ……シ  ┘
```

## リスト 16.4

```
1000 CLEAR 300,&HCFFF
1010 GOSUB 3000
1020 REM MAIN
1030 DEF USR=&HD000
1040 SCREEN 0,,0:CLS:KEY OFF
1050 LOCATE0,1:PRINT" MSX MINI PIANO":PRINT:PRINT  ┐
1060 PRINT"┌──┬──┬──┬──┬──┬──┬──┬──┐   ┌──┐"      │ キーボード
1070 PRINT"│ 1│ 2│ 3│ 4│ 5│ 6│ 7│ 8│   │ 0│"      │ の絵を画面
1080 PRINT"└──┴──┴──┴──┴──┴──┴──┴──┘   └──┘"      ┘ に表示する
1090 PRINT" DO RE MI FA SOLLA TI DO   END"
1100 X=USR(0)
1110 IF INKEY$<>"" THEN 1110
1120 END
3000 REM MACHINE CODE                ┐
3010 FOR I=&HD000 TO &HD076          │
3020  READ D$:D=VAL("&H"+D$)         │ マシンコードの書き込み
3030  POKE I,D                       │
3040 NEXT:RETURN                     ┘
3050 DATA 3E,08.1E.0A,CD.93.00.3E
3060 DATA 07,1E.FF,CD.93,00.3E,00
3070 DATA CD,41.01.CB.47,C8.E6.FE
3080 DATA EE,FE.20.16.3E.01.CD.41
3090 DATA 01,E6,01.28.05,CD.4E.D0
3100 DATA 18,E4,11,01.DD.CD,56.D0
3110 DATA 18,DC.EE.FE.06.07.0E.02
3120 DATA 21,69.D0,5F.A1.7B.28.06
3130 DATA 23,23.CB.01.10.F5.5E.23
3140 DATA 56,CD.56.D0,18.C0.3E,07
3150 DATA 1E,FF.CD.93.00.C9.3E.01
3160 DATA CD,93.00,3E.00.5A.CD.93
3170 DATA 00,3E.07.1E.FE.CD.93,00
3180 DATA C9,03,BA.03.52,02.F5,02
3190 DATA CB,02.7D.02.37.01.F9
```

# 16.3

# ジョイスティックスキャン



MSXには入力装置として，ジョイスティックというものがあります．これは写真のようなもので，主にゲームのコントローラとして使います．ここではプログラムの中で，ジョイスティックを扱ってみましょう．

ジョイスティックはオプション製品ですから，ふつうMSXを買っただけでは付いていません．しかし，キーボードのカーソル移動キーをこのジョイスティックの代わりとして使うことができます．

## ジョイスティックの働き

ジョイスティック(またはカーソル移動キー)は，上下左右にななめを加えた8方向にたおすことができます．MSX-BASICにはSTICK関数というものがあって，これが実行されたときのジョイスティックの状態(どの方向にジョイスティックが向いているか)を調べ，図16.4に示した値を返します．

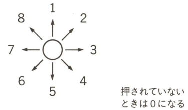



図16.4 ジョイスティックの値

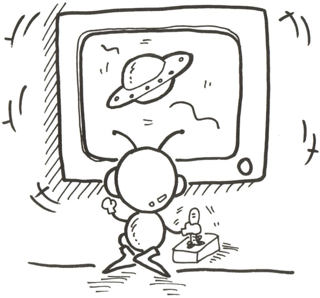
STICK 関数の引数には〝ジョイスティック番号〃というものを指定しますが，これを 0 にすると，カーソル移動キーを調べられます．

マシン語では**表 16.3** の BIOS ルーチンをコールすれば，ジョイスティック(カーソル移動キー)の状態がわかります．

| アドレス | 名　前 | 機　　能 |
|---|---|---|
| 00D5 | GTSTCK | ジョイスティックの状態を調べる<br>●エントリーパラメータ：Aレジスタに調べるジョイスティックの番号を入れる<br>●リターンパラメータ：Aレジスタにジョイスティックが向いている方向がはいる<br>●使用レジスタ：すべて |

**表16.3 ジョイスティックのBIOSエントリー**

A レジスタにジョイスティック番号をロードして，GTSTCK をコールすると，A レジスタにジョイスティックの方向(STICK 関数と同じ)がはいります．ただしこのルーチンはその他のレジスタの内容を保存しないので注意が必要です．



## プログラムのポイント

ここではカーソル移動キーを使って，画面上の文字を動かすプログラムを作ってみましょう．プログラムの中で行うことは，カーソル移動キーの状態を調べて，それによって文字を表示する場所の移動をひたすら繰り返します．これはゲームプログラムのように，キーボードやジョイスティックからの入力を調べて，それによって様々な処理を行うというプログラムの基本になるものです．

この種のプログラムを作る場合には，次のようなことがポイントになります．

- ○ **文字を表示するのとカーソル移動キーを調べるのをどういう順序で行うか**
- ○ **カーソル移動キーの状態によって，どのように分岐するか**
- ○ **文字が画面から出てしまったときにどうするか**

文字を表示する場所は，BIOS の POSIT ルーチンで指定します，この X 座標と Y 座標は，プログラム実行中は変数として扱い，メモリにそのための領域を確保しておきます．そして上に動くときは Y 座標を 1 減らし，右に動くときは X 座標を 1 増やすようにします．

## 文字を移動するプログラム

**リスト 16.5** がマシン語プログラムです．このプログラムを実行する前に，BASIC のプログラムで X，Y(それぞれの座標)には適当な初期値を入れます．

このプログラムではカーソル移動キーの状態を調べ，それによって X と Y の値を増減します．MOVER，MOVEL，MOVEU，MOVED というサブルーチンがそれぞれ座標を右，左，上，下に動かします．ただしそれぞれの方向に動かして画面から出てしまうと困りますから，その場合はカーソル移動キーを押しても文字は動かないようにしなければなりません．

PTSTAR というルーチンは，X，Y という座標に〝*〞という文字を表示しますが，これがコールされるのは，移動によって X 座標，Y 座標が決定した後です．また文字を動かす前に，前の場所にはスペースを表示して，〝*〞を消さなければなりません．

## リスト 16.5

```
D000 CDB700   STCKLP: CALL  BREAKX
D003 D8               RET   C           ; [CTRL]+[STOP]が押されたらBASIC
D004 3E00             LD    A,0         ; に戻る
D006 CDD500           CALL  GTSTCK
D009 2A00D8           LD    HL,(0D800H)
D00C B7               OR    A           ; スティックが押されていな
D00D 28F1             JR    Z,STCKLP    ; ければループに戻る"OR
                                        ; A"は"CP 0"と同じ
D00F FE08             CP    8           ; 図16.4の[8](左上)の方向
D011 2008             JR    NZ,STICK1   ; のとき
D013 CD6BD0           CALL  MOVEL
D016 CD5FD0           CALL  MOVEU
D019 18E5             JR    STCKLP
D01B FE07     STICK1: CP    7           ; [7](左)の方向のとき
D01D 2005             JR    NZ,STICK2
D01F CD6BD0           CALL  MOVEL
D022 18DC             JR    STCKLP
D024 FE06     STICK2: CP    6           ; [6](左下)の方向のとき
D026 2008             JR    NZ,STICK3
D028 CD6BD0           CALL  MOVEL
D02B CD77D0           CALL  MOVED
D02E 18D0             JR    STCKLP
D030 FE05     STICK3: CP    5           ; [5](下)の方向のとき
D032 2005             JR    NZ,STICK4
D034 CD77D0           CALL  MOVED
D037 18C7             JR    STCKLP
D039 FE04     STICK4: CP    4           ; [4](右下)の方向のとき
D03B 2008             JR    NZ,STICK5
D03D CD77D0           CALL  MOVED
D040 CD83D0           CALL  MOVER
D043 18BB             JR    STCKLP
D045 FE03     STICK5: CP    3           ; [3](右)の方向のとき
D047 2005             JR    NZ,STICK6
D049 CD83D0           CALL  MOVER
D04C 18B2             JR    STCKLP
D04E FE02     STICK6: CP    2           ; [2](右上)の方向のとき
D050 2008             JR    NZ,STICK7
D052 CD83D0           CALL  MOVER
D055 CD5FD0           CALL  MOVEU
D058 18A6             JR    STCKLP
D05A CD5FD0   STICK7: CALL  MOVEU       ; [1](上)の方向のとき
D05D 18A1             JR    STCKLP
```

次頁へ続く

```
D05F 7D        MOVEU:  LD    A,L         ┐
D060 FE01              CP    1           │
D062 C8                RET   Z           │
D063 CD8FD0            CALL  PUTSPC      │ キャラクタを上へ動かす
D066 2D                DEC   L           │
D067 CD98D0            CALL  PTSTAR      │
D06A C9                RET               ┘
D06B 7C        MOVEL:  LD    A,H         ┐
D06C FE01              CP    1           │
D06E C8                RET   Z           │
D06F CD8FD0            CALL  PUTSPC      │ キャラクタを左へ動かす
D072 25                DEC   H           │
D073 CD98D0            CALL  PTSTAR      │
D076 C9                RET               ┘
D077 7D        MOVED:  LD    A,L         ┐
D078 FE17              CP    23          │
D07A C8                RET   Z           │
D07B CD8FD0            CALL  PUTSPC      │ キャラクタを下へ動かす
D07E 2C                INC   L           │
D07F CD98D0            CALL  PTSTAR      │
D082 C9                RET               ┘
D083 7C        MOVER:  LD    A,H         ┐
D084 FE28              CP    40          │
D086 C8                RET   Z           │
D087 CD8FD0            CALL  PUTSPC      │ キャラクタを右へ動かす
D08A 24                INC   H           │
D08B CD98D0            CALL  PTSTAR      │
D08E C9                RET               ┘
D08F CDC600    PUTSPC: CALL  POSIT       ┐
D092 3E20              LD    A,20H       │ スペースで"*"を消す
D094 CDA200            CALL  CHPUT       ┘
D097 C9                RET
D098 CDC600    PTSTAR: CALL  POSIT           ┐
D09B 3E2A              LD    A,2AH           │ "*"を表示する
D09D CDA200            CALL  CHPUT           │
D0A0 2200D8            LD    (0D800H),HL     ┘
D0A3 C9                RET
```

## リスト 16.6

```
1000 CLEAR 300,&HCFFF
1010 SCREEN 0
1020 KEY OFF
1030 GOSUB 3000
1040 REM MAIN
1050 POKE &HD800,1:POKE &HD801,1
1060 LOCATE 0,0:PRINT "*"
1070 DEFUSR=&HD000
1080 X=USR(0)
1090 END
3000 REM MACHINE CODE
3010 FOR I=&HD000 TO &HD0A3
3020  READ D$
3030  D=VAL("&H"+D$)
3040  POKE I,D
3050 NEXT
3060 RETURN
3070 DATA CD,B7,00,D8,3E,00,CD,D5
3080 DATA 00,2A,00,D8,B7,28,F1,FE
3090 DATA 08,20,08,CD,6B,D0,CD,5F
3100 DATA D0,18,E5,FE,07,20,05,CD
3110 DATA 6B,D0,18,DC,FE,06,20,08
3120 DATA CD,6B,D0,CD,77,D0,18,D0
3130 DATA FE,05,20,05,CD,77,D0,18
3140 DATA C7,FE,04,20,08,CD,77,D0
3150 DATA CD,83,D0,18,BB,FE,03,20
3160 DATA 05,CD,83,D0,18,B2,FE,02
3170 DATA 20,08,CD,83,D0,CD,5F,D0
3180 DATA 18,A6,CD,5F,D0,18,A1,7D
3190 DATA FE,01,C8,CD,8F,D0,2D,CD
3200 DATA 98,D0,C9,7C,FE,01,C8,CD
3210 DATA 8F,D0,25,CD,98,D0,C9,7D
3220 DATA FE,17,C8,CD,8F,D0,2C,CD
3230 DATA 98,D0,C9,7C,FE,28,C8,CD
3240 DATA 8F,D0,24,CD,98,D0,C9,CD
3250 DATA C6,00,3E,20,CD,A2,00,C9
3260 DATA CD,C6,00,3E,2A,CD,A2,00
3270 DATA 22,00,D8,C9
```

1050: 初期位置(1,1)を，マシン語ルーチンのワークエリアに書き込む

1060: 初期位置(BASICのLOCATEなら(0,0)になる)に"*"を表示する

3000～3050: マシンコードの書き込み

3070～3270: マシンコード

このプログラムを実行すると，文字画面に〝*〞という文字が表示されます．カーソル移動キーを押すと，その方向に文字が動きます．マシン語ですからかなりのスピードで動くのがわかるでしょう．文字画面(40×24文字)では狭すぎるくらいです．

このプログラムはAレジスタにロードする値を変えることにより，動かす文字を変えたり，カーソル移動キーではなく本物のジョイスティックを使うことができます．

# 17章

# 画面操作のテクニック

# 17.1

# VDPとV-RAM

この章ではVDPとV-RAMというものについて見てみます．これは画面に表示する文字などを扱うハードウェアのことで，上手な使い方をすればBASICやBIOSルーチンではできないことがいろいろできるようになります．

MSXではSCREEN命令によって，4種類の画面を選ぶことができます．ここではそのうちスクリーンモード0と1，つまり文字画面の構造を説明するとともに，それを応用したプログラムを作ってみましょう．

## コンピュータの画面

コンピュータを使う上で，〝画面〟というのは非常に重要な役割を果たしています．プログラムは画面を見ながら作りますし，そのプログラムを実行して計算結果を出したりするのにも，画面を使います．コンピュータと人間のコミュニケーションで，最もよく使われるのがこの画面なのです．

MSXにはVDPというLSIがあって，画面をコントロールしています．またVDPは，V-RAM(VIDEO RAM)というメモリに〝いま画面にはどんな文字(または模様)が表示されているか〟ということを記憶させています．

たとえば〝10〟という数字を画面に表示したければ，〝1〟という文字と〝0〟という文字を，このV-RAMに書き込みます．〝RAM〟という名前が付いていることからもわかるように，画面上の文字や絵は自由に書き換えることができます．

V-RAMはCPUにつながっているメモリとは別で，VDPによって管理されています．図17.1はMSXが画面に文字や絵を表示するしくみを示したものです．

BASICのPRINT文や，BIOSルーチンのCHPUTなどが実行されると，CPUはVDPを通してV-RAMにその文字の情報を書き込みます．VDPはそのV-RAMを

読み取って，それを画面信号に変換し，テレビやCRTスクリーンの画面に表示しています．

このVDPがV-RAMから情報を読み出して画面信号を出すという操作は，CPUの動きとは関係なく随時行われます．

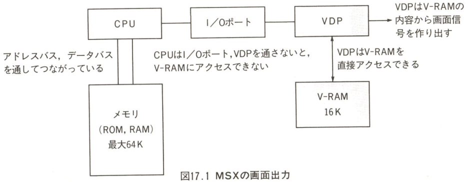



図17.1 MSXの画面出力

## 画面を扱う方法

画面に文字や絵を表示しようと思ったら，このV-RAMにデータを書き込んでやればよいのです．しかし，図17.1にあるように，CPUとV-RAMは直接つながっていませんから，ロード命令で書き込みを行うことはできません．V-RAMにデータを書き込むには，VDPに対して〝何番地に何を書き込め〟という命令を送ってやらなければなりません．

CPUとVDPはI／Oポートを通してつながっていますから，V-RAMへの書き込みはポートを介して操作することになります．VDPのポート番号などを調べれば，I／Oポートを使って書き込みを行うプログラムを作ることができます．しかしこの操作をもっと簡単に行うBIOSルーチンが用意されていますから，ここではこれを使うことにします．

V-RAMをアクセスするBIOSルーチンには，表17.1のようなものがあります．

RDVRMとWRTVRMが，V-RAMに対する読み書きを行うもので，BASICでいうPEEK関数とPOKE文にあたります．これらをコールするときはHLレジスタに

V-RAMのアドレスを指定しますが，この2つがあれば，V-RAMを自由にアクセスすることができます．

そのほかの3つは，一度に大量のV-RAMのデータを操作するものです．ブロック転送命令と同様に,うまく使えば便利なものです．FILVRMは一度に大量のV-RAMに同じデータを書き込むというものですし，LDIRMVとLDIRVMは，V-RAMとCPUにつながっているメモリとの間でブロック転送を行うものです．

LDIRVMとLDIRMVは，V-RAMの内容を一度にたくさん扱いたいときに使います．LDIRMVでV-RAMの内容をメモリ上に持ってくれば，画面のデータがメモリの上で扱えます．メモリ上でデータの書き込みが終わったら，それをLDIRVMでV-RAMにブロック転送すれば，そのデータが画面に表示されるわけです．

| アドレス | 名　前 | 機　　能 |
|---|---|---|
| 004A | RDVRM | V-RAMの指定したアドレスの値を読む<br>●エントリーパラメータ：HLで読み込む番地を指定する<br>●リターンパラメータ　：Aレジスタに読み込んだ値がはいる<br>●使用レジスタ　　　　：AF |
| 004D | WRTVRM | V-RAMの指定したアドレスに値を書き込む<br>●エントリーパラメータ：HLレジスタに書き込むアドレスを入れ，Aレジスタに書き込む値を入れる<br>●リターンパラメータ　：なし<br>●使用レジスタ　　　　：AF |
| 0056 | FILVRM | V-RAMを指定した値でうめる<br>●エントリーパラメータ：HLレジスタに書き込むV-RAMの先頭アドレス，Aレジスタに書き込む値，BCレジスタに値をいくつ書き込むかを指定する<br>●リターンパラメータ　：なし<br>●使用レジスタ　　　　：AF, BC |
| 0059 | LDIRMV | V-RAMからメモリへブロック転送する<br>●エントリーパラメータ：HLレジスタに転送元の先頭アドレス，DEレジスタに転送先の先頭アドレス，BCレジスタに転送するバイト数を指定する<br>●リターンパラメータ　：なし<br>●使用レジスタ　　　　：すべて |
| 005C | LDIRVM | メモリからV-RAMへブロック転送する<br>●エントリーパラメータ：HLレジスタに転送元の先頭アドレス，DEレジスタに転送先の先頭アドレス，BCレジスタに転送するバイト数を指定する<br>●リターンパラメータ　：なし<br>●使用レジスタ　　　　：すべて |

表17.1 VDP関係のBIOS

# 17.2

# V-RAMの構造

VDP関係のBIOSルーチンを利用すれば，画面に文字や絵を表示させることはできるのですが，その前に〝何番地にどんなデータを書き込めば，どういう画面が表示されるか〟ということを知っておく必要があります．これが〝V-RAMの構造〟というものです．MSXのVDPでは，画面はスクリーンモード0から3まで4種類の画面があるため，それぞれのモードではこのV-RAMの構造が変わってきます．

文字画面(スクリーンモード0と1)の構造は次の項で説明しますが，これをしっかりと理解して使えるようになれば，画面に文字を好きなように表示することができます．同様にグラフィック画面についてもV-RAMの構造を理解すれば，点や直線を描くことができます．しかしグラフィック画面についてはこの本では扱っていません．

## 文字画面を扱う

MSXのVDPは16KのV-RAMを持っています．その構造はCPUに付いているメモリと同じで，1つのアドレスに1バイトのデータがはいります．アドレスは16K分，つまり0Hから4000Hまで指定できます．

文字画面として使われるときは，このV-RAMをいくつかに分割しますが，大切なものは〝パターン名称テーブル〟と〝パターンジェネレーターテーブル〟の2つです．

V-RAMを文字画面に使うときは，〝どのキャラクタコードの文字を表示するか〟と，〝どんな形の文字を表示するか〟がV-RAMに記憶される情報になります．前者がパターン名称テーブルで，後者がパターンジェネレーターテーブルです．ほかにもスプライトに関するものなどがありますが，この章ではこの2つがわかれば十分です．

BASICでもこのV-RAMをアクセスする命令があります．VPEEK関数とVPOKE文がそうです．これらはちょうどPEEK関数，POKE文と同じように使うこ

とができますから，実際にV-RAMの構造を調べるときには，これら2つの命令を使って試してみるとよいでしょう.

## スクリーンモード0のパターン名称テーブル

スクリーンモード0はテキスト画面と呼ばれ，文字だけを表示することができるモードです．ただしこの画面では1つの文字を横6ドットで表示するため，ひらがなを表示すると右側が欠けてしまいます．この画面は横40×縦24文字という大きさですが，960個あるマス目のそれぞれに，1バイトが対応したものをパターン名称テーブルといいます.

これは図17.2のように0番地から959(3BFH)番地までが割り当てられていて，それぞれ画面の各座標と対応しています.

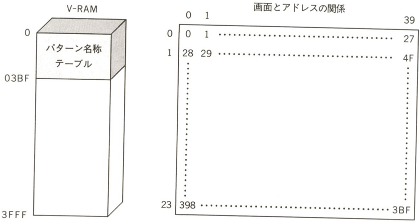



図17.2 スクリーンモード0のパターン名称テーブルのアドレス

それぞれのアドレスにキャラクタコードを書き込めば，その座標にそのコードに対応する文字が表示されるようになっています.試しにいくつか書き込んでみましょう.スクリーンモードは0にしておいてください.

```
VPOKE &H27,&H58
Ok
■
```

```
VPOKE &H3BF,&H25
Ok
■
```

左の例は画面の(39，0)の場所に〝X″(キャラクタコードは５８Ｈ)という文字を表示するものです．また，右の例では画面の右下に〝%″が表示されます．このアドレス(３ＢＦＨ)はスクリーンモード０の右下の座標です．ふつうこの座標に何か文字を書くと，自動的にスクロールが起こってしまい，書いた文字は１行上にいってしまうので，PRINT 文(CHPUT ルーチン)ではこの座標に文字を書くことはできません．しかし VPOKE 文を使うと，ただ V-RAM に書き込みを行うだけでスクロールは起こりませんから，この場所に文字を表示することができるのです．

文字画面にグラフィックキャラクタを書きたいときは，０Ｈから１ＦＨまでのコードを書き込みます．次の例では〝日″という文字が表示されます．

```
VPOKE &H27,&H7                                    日
Ok
```

BIOS ルーチンの CHPUT で７Ｈというコードを表示しようとすると，ビープ音が出ます．これはキャラクタコード０Ｈから１ＦＨまでは，コントロールコードといって，特殊な働きをするものとして使われているからです．

CHPUT でグラフィックキャラクタを表示するには，まずコード１を出力して，それからグラフィックキャラクタのコードを表示するというように，２文字分の操作が必要になります．

```
PRINT CHR$(1)+CHR$(&H47)
日
Ok
```

こうすると CHPUT ルーチンの中で，V-RAM に０Ｈから１ＦＨまでのコードを書き込んでくれるのです．しかし V-RAM に直接書き込みを行うときは，ほかのふつうの文字と同様１文字(１バイト)として扱うことができます．

## パターンジェネレーターテーブル

次はパターンジェネレーターテーブルです．これは〝0HからFFHまでのそれぞれの文字に対し，どのような形を画面に出すか″ということを書いておくものです．たとえばパターン名称テーブルに41Hというコードを書き込むと，〝A″という文字が画面に表示されますが，これは〝A″という文字の形が，パターンジェネレーターテーブルに書いてあるからできるのです．

パターンジェネレーターテーブルは，図17.3のように0HからFFHまで256文字分のデータを貯えているメモリです．1つの文字のパターンは8ドット×8ドットでできていますから，1文字につき8バイト(8ドットが1バイトになる)，全部で2048(800H)バイトです．

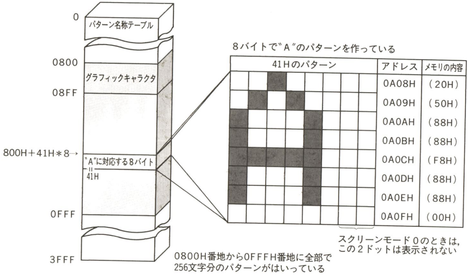



**図17.3 スクリーンモード0のパターンジェネレータテーブル**

スクリーンモード0のパターンジェネレーターテーブルは800H番地から，FFFH番地までが割り当てられています．**リスト17.1**は各文字がどのようなパターンになっているかを調べるプログラムです．

## リスト 17.1

```
100 REM PATTERN GENERATOR TABLE
110 INPUT "CHARACTER";A$        } 文字を入力して，そのキャラクタコードを求める
120 C=ASC(A$)                   }
130 VA=&H800+C*8 ……………文字パターンのアドレスを求める
140 FOR I=VA TO VA+7 ……………1文字について8バイト
150 P=VPEEK(I)
160 PRINT RIGHT$("0000000"+BIN$(P),8)
170 NEXT
```

リスト 17.1 を実行すると，入力した文字のパターンを読み出して，それを 2 進数で表示しますから，文字パターンを大きくしてみることができます．図 17.4 はその実行例です．

```
run ↵
CHARACTER? F ↵
11111000 )
10000000 |
10000000 |
11110000 | "F"という文字のパターン
10000000 |
10000000 |
10000000 |
00000000 )
Ok
█
```

図17.4 リスト17.1実行例

ところでこのパターンジェネレーターテーブルは，MSX をパワーオンしたときに 256 種類の文字パターンがすべて書き込まれます．しかしここは RAM ですから，勝手に書き換えて自分の好きな形の文字を表示させるということもできます．ただし，SCREEN 文を実行して画面を設定し直すと，またパワーオン時のパターンが書き込まれるので，せっかく書き込んだパターンがなくなってしまいますから注意してください．

書き換えるときは，リスト 17.1 の 130 行目と同じ式でアドレスを求めることができます．たとえば変えたい文字のコードが C という変数にはいっていたら，

VA=&H800+C*8

からの8バイトに，VPOKE文で好きなパターンを書き込むことで可能になります.

## スクリーンモード1

スクリーンモード1は0と同じく文字だけを表示する画面です．この画面では，横32×縦24になりますから，合計768文字しか表示することができません．しかし1つの文字は横8ドットで表現されるので，スクリーンモード0のようにひらがなの横が切れるということはありません.

まずパターン名称テーブルは図17.5に示すように1800H番地からの768バイト(300H)にあります．この画面のX，Y座標からアドレスを求めるには次のようにします.

VA=&H1800+X+Y*32

またパターンジェネレーターテーブルは0番地から始まっていますから，キャラクタコードCのパターンはC*8番地からの8バイトにはいっていることになります.

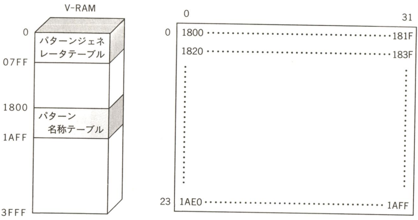



図17.5 スクリーンモード1のパターン名称テーブル

# 17.3

# 文字の回転

この節では，パターンジェネレーターテーブルを扱って，文字が回転するというプログラムを作ってみましょう．これはシフト命令を使って，8ドット×8ドットの文字パターンを横にしてみようというものです．

## プログラムの手順

このプログラムでは次のようなことを行います．

**① V-RAM からパターンジェネレーターテーブルの1文字分(8バイト)のデータをメモリに持ってくる**
**② シフト命令を使って回転したパターンを得る(これは1バイトずつ)**
**③ その回転したパターンを V-RAM に書き込む**

どの文字のパターンを扱うかということは，BASIC の INPUT 文で入力して，それを USR 関数の引数としてマシン語プログラムに受け渡すことにしましょう．

文字の回転は**図 17.6** のようにして行います．

1つの文字は8ドット×8ドットのパターンでできていますが，**図 17.6** のように連続した8バイトにはいっています．この8バイトのビット7をすべて取り出して A レジスタのビット0からビット7に入れると，これはもとのパターンの縦8ビットが取れたことになります．これをパターンジェネレーターテーブルに書き込むと，横8ビットとして表示されます．ですからこれをもとのパターンのビット6からビット0まで，すべてについて行えば，8ドット×8ドットのパターンが横になるということはわかりますね．

ビットを1つずつ取り出して A レジスタに入れるところは，シフト命令とローテイ

ト命令を組み合せます．これは（HL）のビット7をCYフラグを通して，Aレジスタのビット0に送るというものです．

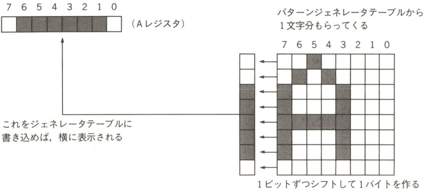



図17.6 横にする方法

## 文字回転プログラム

このプログラムは，スクリーンモード1で行います．なぜかというと，8ドット×8ドットのパターンすべてが画面に表示されないと，回転している感じが出ないからです．

**リスト17.2**が，マシン語のプログラムです．さきほどと同様に，このプログラムにもBASICの部分とマシン語の部分があります．BASICの部分では回転させたい文字を入力して，そのキャラクタコードからV-RAMのパターンジェネレーターテーブルのアドレスを求め，それをマシン語ルーチンに渡します．ですから，マシン語ルーチンでは渡されたV-RAMのアドレスからの8バイトを回転させればよいのです．



## リスト 17.2

```
D000 23                   INC     HL
D001 23                   INC     HL
D002 5E                   LD      E,(HL)   BASICからUSR関数の引数と
D003 23                   INC     HL       して渡されたパターンジェネレ
D004 56                   LD      D,(HL)   ータテーブルのアドレスを,
D005 62                   LD      H,D      HLレジスタに入れる
D006 6B                   LD      L,E
D007 E5          ROT:     PUSH    HL
D008 010800               LD      BC,8
D00B 112CD0               LD      DE,0D02CH
D00E D5                   PUSH    DE
D00F CD5900               CALL    LDIRMV   ……1文字分のパターン(8バイト)
D012 D1                   POP     DE         をメモリに移す
D013 E1                   POP     HL
                          ;
D014 0608                 LD      B,8
D016 C5          ROT1:    PUSH    BC
D017 0608                 LD      B,8
D019 D5                   PUSH    DE
D01A EB                   EX      DE,HL
D01B CB26        ROT2:    SLA     (HL)     メモリから1ビットずつ取り出して
D01D CB1F                 RR      A        Aレジスタに入れる
D01F 23                   INC     HL
D020 10F9                 DJNZ    ROT2
                          ;
D022 EB                   EX      DE,HL
D023 CD4D00               CALL    WRTVRM
D026 23                   INC     HL
D027 D1                   POP     DE
                          ;
D028 C1                   POP     BC
D029 10EB                 DJNZ    ROT1
                          ;
D02B C9   …D02CH番地からD030H番  RET
           地にワークエリアを取る
D02C 00000000
D030 00000000
```

このプログラムの中では，最初に BASIC から渡されるパターンジェネレーターテーブル(V-RAM)のアドレスと，それをコピーしてきたメモリのアドレスを覚えていなければなりません．これらはそれぞれ HL, DE レジスタで扱っていますが，パターンを横にする部分でそれを入れ替えています．

**リスト 17.3** は**リスト 17.2** をコールする BASIC のプログラムです．これはまず文

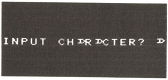
字を入力して，その文字のパターンジェネレーターテーブルのアドレスを求めます．リストの1080行目では，グラフィックキャラクタが入力された場合のパターンジェネレーターテーブルのアドレスを求めています．このアドレスをUSR関数の引数としてマシン語に渡しますが，これは必ず整数型で扱っておかなければなりません．

マシン語プログラムを1回コールすると，文字が90度右に回ります．ですから何回もマシン語をコールすれば，文字が回転するように見えるのです．

## リスト 17.3

```
1000 CLEAR 300,&HCFFF                              初期設定
1010 SCREEN 1
1020 DEFINT A
1030 DEF USR=&HD000
1040 GOSUB 3000
1050 REM MAIN
1060 INPUT "INPUT CHARACTER";C$
1070 IF ASC(C$)>1 THEN 1100
1080 A=(ASC(RIGHT$(C$,1))-&H40)*8 ……………グラフィックキャラクタの場合
1090 GOTO 1110
1100 A=ASC(C$)*8 ……………パターンジェネレータテーブルのアドレス
1110 X=USR(A)
1120 FOR I=0 TO 100:NEXT        時間待ちをして無限にマシン語をコールする
1130 GOTO 1110
3000 REM MACHINE CODE
3010 FOR I=&HD000 TO &HD033                       マシンコードの書き込み
3020  READ D$:D=VAL("&h"+D$)
3030  POKE I,D
3040 NEXT
3050 RETURN
3060 DATA 23,23,5E,23,56,62,6B,E5
3070 DATA 01,08,00,11,2C,D0,D5,CD
3080 DATA 59,00,D1,E1,06,08,C5,06
3090 DATA 08,D5,EB,CB,26,CB,1F,23
3100 DATA 10,F9,EB,CD,4D,00,23,D1
3110 DATA C1,10,EB,C9,00,00,00,00
3120 DATA 00,00,00,00
```







このプログラムを実行すると，まず文字を尋ねてきます．そこで1文字入力しますと，その文字が回転します．

# 17.4

# ライフゲーム

ここではパターン名称テーブルを扱う例として，ライフゲームを作ってみます．このプログラムでもV-RAMの内容を，メモリで扱いますが，今度は逆にLDIRVMを使ってメモリからV-RAMにブロック転送を行います．

## ライフゲームについて

ライフゲーム(Life Game)というのは，あるシステム(生物の群れをシミュレートしたものといわれている)が一定のルールに従って変化していくというもので，コンピュータの世界では有名なものです．〝ゲーム〟という名前がついていますが，残念ながらいわゆる遊ぶためのゲームではありません．画面に適当なドットの集まりのパターンを描いて，ルールに従ってそのパターンを変化させていくという視覚的なプログラムです．ライフゲームは次のルールに従って行います．

**① 画面上の全ドットは，生きているか死んでいるかのいずれかである**

**② 全ドットについて，そのまわりの8つのドットの状態を調べる(図17.7)**

**③ 8ドットのうち生きているドットの数によって，次のようにそのドットの生死が決定する**

| 生きていたドットの数 | 次の世代の生死 |
|---|---|
| 0～1 | 死ぬ |
| 2 | そのまま |
| 3 | 生まれる(生きていた場合はそのまま) |
| 4以上 | 死ぬ |

**④ ③のルールに従って各ドットの次の状態が決まると，次の世代に進む**

ただし画面の端のものについては，それが画面の反対側とつながっていることになっています．ですからいちばん上のドットの上というのは，画面のいちばん下のドットになります．

ライフゲームで肝心なのは，画面上の全ドットが一度に次の世代に替わるということです．ですから図 17.7 の e が f を調べて次の状態が決まったからといって，その時点で状態が変わってはいけません．なぜなら f が e を調べるときの e の状態は，e が f を調べているときの状態でなければならないからです．

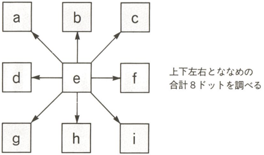



図17.7 ライフゲーム

そのためライフゲームのプログラムでは，画面の状態と同じものをどこか別の場所にとっておかなければなりません．今回は V-RAM と同じものを，メモリ上にとっておきます．V-RAM のドット（文字画面なのでキャラクタを使う）を調べて次の状態が決まったら，それをメモリの方に書き込んでおき，そして全部のドットについて生死が決まったら，メモリにあるものをいっぺんに画面（V-RAM）に書き込みます．これで世代が 1 つ進むのです．

**リスト 17.4** がライフゲームのマシン語部分，**リスト 17.5** が BASIC とマシン語を結合したプログラムです．ライフゲームの主な部分はすべてマシン語で行います．このプログラムではさきほどの②，③のルールに従って，各キャラクタの生死を判定して 1 世代の画面を作ります．生きている文字は〝*″，死んだ文字はスペースが表示されます．

**リスト 17.5** の BASIC の部分は，マシン語をメモリに入れる部分と初期画面を作る部分から成っています．このうち，初期画面を作るのは 2000 行目からです．プログラ

ムを実行した時点では，画面はクリアされ，左上端にカーソルが表示されています．ここでカーソルを動かしてスペースキーを押すと，その場所に〝*〟が表示され，これが生きているドットを表すことになります．

ここでリターンキーを押せばライフゲームが始まり，スペースキーでプログラムが終了します．マシン語プログラムを1回コールすると，世代が1つ進みますから，これを何度もコールすると，世代がどんどん進んでいくように見えるのです．

## リスト 17.4

```
D000 0628     LIFE:   LD     B,40   …X座標（ループの中   最初はともに
D002 48       LIFE1:  LD     C,B       ではCレジスタ      自分の値より
D003 C5               PUSH   BC                          1多い
D004 0618             LD     B,24   ……Y座標
D006 C5       LIFE2:  PUSH   BC
D007 50               LD     D,B
D008 15               DEC    D              ……Y座標は自分の値に戻す
D009 59               LD     E,C
D00A 0600             LD     B,0            ……自分のまわりの生きている個数
D00C CD77D0           CALL   CHECKX   ……右を見る
D00F 14               INC    D              ……Y座標を1増やして下に移動する
D010 CD77D0           CALL   CHECKX   ……右下を見る
D013 1D               DEC    E              ……X座標を1減らして左に移動する
D014 CD77D0           CALL   CHECKX   ……下を見る
D017 1D               DEC    E
D018 CD77D0           CALL   CHECKX   ……左下を見る
D01B 15               DEC    D
D01C CD77D0           CALL   CHECKX   ……左を見る
D01F 15               DEC    D
D020 CD77D0           CALL   CHECKX   ……左上を見る
D023 1C               INC    E
D024 CD77D0           CALL   CHECKX   ……上を見る
D027 1C               INC    E
D028 CD77D0           CALL   CHECKX   ……右上を見る
D02B 1D               DEC    E
D02C 14               INC    D  ……自分の位置に戻す
D02D CD7FD0           CALL   VRAMAD
D030 CD4A00           CALL   RDVRM
D033 4F               LD     C,A  ……今の状態(スペースか*か)
D034 78               LD     A,B  …CHECKXによって求められた生きてい
D035 FE02             CP     2       る数がBレジスタにはいっているので，
                                     それをAレジスタに移す
D037 2005             JR     NZ,LIFE3   まわりの生きている数が2な
D039 CD66D0           CALL   STAY       ら"STAY"をコールする
D03C 180C             JR     LIFE5
```

次頁へ続く

```
D03E FE03       LIFE3:  CP      3
D040 2005               JR      NZ,LIFE4
D042 CD5DD0             CALL    BIRTH   ……まわりの生きている数が3なら
D045 1803               JR      LIFE5     "BIRTH"をコールする
D047 CD6ED0     LIFE4:  CALL    DEATH   ……それ以外のとき(0~1,4以上)は
D04A C1         LIFE5:  POP     BC        "DEATH"をコールする
D04B 10B9               DJNZ    LIFE2   ……内側のループ(Y座標)
D04D C1                 POP     BC
D04E 10B2               DJNZ    LIFE1   ……外側のループ(X座標)
D050 2100D8             LD      HL,0D800H ┐
D053 110000             LD      DE,0      │ メモリの状態を一度に
D056 01C003             LD      BC,960    │ V-RAMに書き込む
D059 CD5C00             CALL    LDIRVM    ┘
D05C C9                 RET
                        :
D05D D5         BIRTH:  PUSH    DE        ┐
D05E 1100D8             LD      DE,0D800H │
D061 19                 ADD     HL,DE     │ 生まれる
D062 362A               LD      (HL),2AH  │
D064 D1                 POP     DE        ┘
D065 C9                 RET
D066 D5         STAY:   PUSH    DE        ┐
D067 1100D8             LD      DE,0D800H │
D06A 19                 ADD     HL,DE     │ そのまま
D06B 71                 LD      (HL),C    │
D06C D1                 POP     DE        ┘
D06D C9                 RET
D06E D5         DEATH:  PUSH    DE        ┐
D06F 1100D8             LD      DE,0D800H │
D072 19                 ADD     HL,DE     │ 死ぬ
D073 3620               LD      (HL),20H  │
D075 D1                 POP     DE        ┘
D076 C9                 RET
                        :
D077 CD91D0     CHECKX: CALL    CHECK     ┐
D07A FE20               CP      20H       │ 状態を見て，生きていたらBレジス
D07C C8                 RET     Z         │ タと1増やす
D07D 04                 INC     B         │
D07E C9                 RET               ┘
                        :
D07F D5         VRAMAD: PUSH    DE        ┐
D080 7A                 LD      A,D       │
D081 CB27               SLA     A         │
D083 CB27               SLA     A         │ V-RAMのアドレスを求める
D085 82                 ADD     A,D       │ エントリーパラメータ
D086 2600               LD      H,0       │   Dレジスタ…Y座標
D088 6F                 LD      L,A       │   Eレジスタ…X座標
D089 29                 ADD     HL,HL     │ リターンパラメータ
                                          │   HLレジスタ…V-RAMのアドレス
```

次頁へ続く

```
D08A 29                  ADD    HL,HL
D08B 29                  ADD    HL,HL
D08C 1600                LD     D,0
D08E 19                  ADD    HL,DE
D08F D1                  POP    DE
D090 C9                  RET
                         ;
D091 D5         CHECK:   PUSH   DE
D092 7A                  LD     A,D
D093 FEFF                CP     0FFH
D095 2002                JR     NZ,CHECK1
D097 1617                LD     D,23
D099 FE18       CHECK1:  CP     24
D09B 2002                JR     NZ,CHECK2
D09D 1600                LD     D,0
D09F 7B         CHECK2:  LD     A,E
D0A0 FEFF                CP     0FFH
D0A2 2002                JR     NZ,CHECK3
D0A4 1E27                LD     E,39
D0A6 FE28       CHECK3:  CP     40
D0A8 2002                JR     NZ,CHECK4
D0AA 1E00                LD     E,0
D0AC CD7FD0     CHECK4:  CALL   VRAMAD
D0AF CD4A00              CALL   RDVRM
D0B2 D1                  POP    DE
D0B3 C9                  RET
```

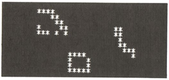
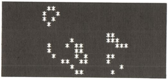
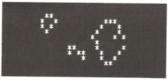
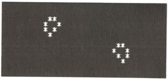
座標からV-RAMアドレスを求めて，状態を調べる．ただしX座標，Y座標が画面から出てしまったときは，反対側を見る

## リスト 17.5

```
1000 CLEAR 300.&HCFFF
1010 DEFINT A
1020 SCREEN 0.,0:KEY OFF:WIDTH 40:CLS
1030 DEFUSR=&HD000
1040 GOSUB 3000
1050 GOSUB 2000
1060 X=USR(0)
1070 IF INKEY$=" " THEN END
1080 GOTO 1060
2000 REM INITIAL STATE  ……………初期画面の設定
2010 CLS:DIM A(39.23)
2020 X=0:Y=0
2030 LOCATE X.Y.1
2040 IF STRIG(0) THEN GOSUB 2140 …………スペースキーが押されたらカーソルの場所の状態を反転する
2050 S=STICK(0)
```

次頁へ続く

```
2060 IF S=1 AND Y>0 THEN Y=Y-1
2070 IF S=7 AND X>0 THEN X=X-1        カーソルの移動
2080 IF S=5 AND Y<23 THEN Y=Y+1
2090 IF S=3 AND X<39 THEN X=X+1
2100 IF INKEY$=CHR$(13) THEN 2130 …………リターンキーが押されたら終了する
2110 FOR I=0 TO 40:NEXT
2120 GOTO 2030
2130 LOCATE ,,0:RETURN
2140 A=A(X,Y)
2150 IF A THEN A(X,Y)=0 ELSE A(X,Y)=1
2160 LOCATE (X XOR 1),Y              カーソルの場所の生死を反転する
2170 VPOKE X+Y*40,32+A*10
2180 LOCATE X,Y,1
2190 RETURN
3000 REM MACHINE CODE
3010 FOR I=&HD000 TO &HD0B3
3020  READ D$:D=VAL("&H"+D$)         マシンコードの書き込み
3030  POKE I,D
3040 NEXT
3050 RETURN
3060 DATA 06,28,48,C5,06,18,C5,50
3070 DATA 15,59,06,00,CD,77,D0,14
3080 DATA CD,77,D0,1D,CD,77,D0,1D
3090 DATA CD,77,D0,15,CD,77,D0,15
3100 DATA CD,77,D0,1C,CD,77,D0,1C
3110 DATA CD,77,D0,1D,14,CD,7F,D0
3120 DATA CD,4A,00,4F,78,FE,02,20
3130 DATA 05,CD,66,D0,18,0C,FE,03
3140 DATA 20,05,CD,5D,D0,18,03,CD
3150 DATA 6E,D0,C1,10,B9,C1,10,B2
3160 DATA 21,00,D8,11,00,00,01,C0
3170 DATA 03,CD,5C,00,C9,D5,11,00
3180 DATA D8,19,36,2A,D1,C9,D5,11
3190 DATA 00,D8,19,71,D1,C9,D5,11
3200 DATA 00,D8,19,36,20,D1,C9,CD
3210 DATA 91,D0,FE,20,C8,04,C9,D5
3220 DATA 7A,CB,27,CB,27,82,26,00
3230 DATA 6F,29,29,29,16,00,19,D1
3240 DATA C9,D5,7A,FE,FF,20,02,16
3250 DATA 17,FE,18,20,02,16,00,7B
3260 DATA FE,FF,20,02,1E,27,FE,28
3270 DATA 20,02,1E,00,CD,7F,D0,CD
3280 DATA 4A,00,D1,C9
```

# 付録

# BIOSエントリー表

ここに掲載したBIOSルーチンは，MSXのBIOSルーチンとして公開されているもののうち，有用と思われるものを抜すいしたものです．

## ■VDP(TMS9918A)に対する入出力ルーチン

| アドレス | 名　前 | 機　　能 |
|---|---|---|
| 0041 | DISSCR | 画面の表示を停止する<br>●エントリーパラメータ：なし<br>●リターンパラメータ　：なし<br>●使用レジスタ　　　　：AF, BC |
| 0044 | ENASCR | 画面の表示を開始する<br>●エントリーパラメータ：なし<br>●リターンパラメータ　：なし<br>●使用レジスタ　　　　：AF, BC |
| 0047 | WRTVDP | 指定したVDPレジスタに値を書き込む<br>●エントリーパラメータ：<br>C←VDPのレジスタ番号<br>B←書き込む値<br>●リターンパラメータ　：なし<br>●使用レジスタ　　　　：AF, BC |
| 004A | RDVRM | V-RAM(ビデオRAM)の指定したアドレスの値を読む<br>●エントリーパラメータ：<br>HL←読み込むアドレス<br>●リターンパラメータ　：<br>A←読み込んだ値<br>●使用レジスタ　　　　：AF |
| 004D | WRTVRM | V-RAMの指定したアドレスに値を書き込む<br>●エントリーパラメータ：<br>HL←書き込むアドレス<br>A←書き込む値<br>●リターンパラメータ　：なし<br>●使用レジスタ　　　　：AF |
| 0050 | SETRD | VDPを読み込み状態にする<br>●エントリーパラメータ：HL<br>●リターンパラメータ　：なし<br>●使用レジスタ　　　　：AF |
| 0053 | SETWRT | VDPを書き込み状態にする<br>●エントリーパラメータ：HL<br>●リターンパラメータ　：なし<br>●使用レジスタ　　　　：AF |

| アドレス | 名　前 | 機　　能 |
|---|---|---|
| 0056 | FILVRM | V-RAMを指定した値で埋める<br>●エントリーパラメータ：<br>HL←書き込むV-RAMの先頭アドレス<br>A←書き込む値<br>BC←書き込むバイト数<br>●リターンパラメータ　：なし<br>●使用レジスタ　　　　：AF, BC |
| 0059 | LDIRMV | V-RAMからメモリヘブロック転送する<br>●エントリーパラメータ：<br>HL←転送元の先頭アドレス<br>DE←転送先の先頭アドレス<br>BC←転送するバイト数<br>●リターンパラメータ　：なし<br>●使用レジスタ　　　　：すべて |
| 005C | LDIRVM | メモリからV-RAMヘブロック転送する<br>●エントリーパラメータ：<br>HL←転送元の先頭アドレス<br>DE←転送先の先頭アドレス<br>BC←転送するバイト数<br>●リターンパラメータ　：なし<br>●使用レジスタ　　　　：すべて |
| 0084 | CALPAT | スプライト・ジェネレータ・テーブルのアドレスを求める<br>●エントリーパラメータ：<br>A←求めるスプライト番号<br>●リターンパラメータ　：<br>HL←求めるスプライト・ジェネレータ・テーブルのアドレス<br>●使用レジスタ　　　　：AF, DE, HL |
| 0087 | CALATR | スプライト属性テーブルのアドレスを求める<br>●エントリーパラメータ：<br>A←求めるスプライト番号<br>●リターンパラメータ　：<br>HL←求めるスプライト属性テーブルのアドレス<br>●使用レジスタ　　　　：AF, DE, HL |

| アドレス | 名　前 | 機　　能 |
|---|---|---|
| 008A | GSPSIZ | 現在のスプライトサイズを求める<br>●エントリーパラメータ：なし<br>●リターンパラメータ　：<br>A←スプライトサイズ<br>16×16のスプライトサイズを使っていたときのみ，CYフラグが立つ<br>●使用レジスタ　　　：AF |
| 008D | GRPPRT | グラフィック画面に文字を表示する<br>●エントリーパラメータ：<br>A←キャラクタコード<br>●リターンパラメータ　：なし<br>●使用レジスタ　　　：なし |

## ■PSGへの入出力ルーチン

| アドレス | 名　前 | 機　　能 |
|---|---|---|
| 0090 | GICINI | PSGを初期化して，PLAY文用の定数をセットする<br>●エントリーパラメータ：なし<br>●リターンパラメータ　：なし<br>●使用レジスタ　　　：すべて |
| 0093 | WRTPSG | PSGレジスタに値を出力する<br>●エントリーパラメータ：<br>A←PSGレジスタの番号<br>E←出力する値<br>●リターンパラメータ　：なし<br>●使用レジスタ　　　：なし |
| 0096 | RDPSG | PSGレジスタの値を読む<br>●エントリーパラメータ：<br>A←PSGレジスタの番号<br>●リターンパラメータ　：<br>A←PSGレジスタから入力した値<br>●使用レジスタ　　　：なし |
| 0099 | STRTMS | PLAY文のバックグラウンドタスクをチェックして開始する<br>●エントリーパラメータ：なし<br>●リターンパラメータ　：なし<br>●使用レジスタ　　　：すべて |

## ■キーボードやディスプレイに対する入出力ルーチン

| アドレス | 名　前 | 機　　能 |
|---|---|---|
| 009C | CHSNS | キーボードバッファの状態を調べる<br>●エントリーパラメータ：なし<br>●リターンパラメータ　：<br>バッファに文字がはいっていれば，Zフラグをリセットする<br>●使用レジスタ　　　：AF |
| 009F | CHGET | 入力があるまで待ち，入力したキャラクタコードを求める<br>●エントリーパラメータ：なし<br>●リターンパラメータ　：<br>A←入力したキャラクタコード<br>●使用レジスタ　　　：AF |
| 00A2 | CHPUT | 画面に，指定した文字を表示する<br>●エントリーパラメータ：<br>A←キャラクタコード<br>●リターンパラメータ　：なし<br>●使用レジスタ　　　：なし |
| 00A5 | LPTOUT | プリンタに，指定した文字を出力する<br>●エントリーパラメータ：<br>A←キャラクタコード<br>●リターンパラメータ　：<br>異常終了したらCYフラグを立てる<br>●使用レジスタ　　　：F |
| 00A8 | LPTSTT | プリンタの状態をチェックする<br>●エントリーパラメータ：なし<br>●リターンパラメータ　：<br>A＝255でかつ，Zフラグが立っていなければプリンタを使える状態である<br>A＝0でかつ，Zフラグが立っていればプリンタは使えない<br>●使用レジスタ　　　：AF |
| 00AB | CNVCHR | グラフィックヘッダバイトをチェックしコード変換する<br>●エントリーパラメータ：<br>A←チェックするキャラクタコード<br>●リターンパラメータ　：<br>CYフラグが立っていない　A←グラフィックヘッダバイト<br>CYフラグとZフラグが立っている<br>A←変換後の値<br>CYフラグが立ちZフラグが立っていない　A←変換しないキャラクタコード<br>●使用レジスタ　　　：AF |

| アドレス | 名　前 | 機　　能 |
|---|---|---|
| 00AE | PINLIN | キャリッジリターンやSTOPを入力するまでに入れた値をラインバッファに入れる<br>●エントリーパラメータ：なし<br>●リターンパラメータ　：<br>HL←ラインバッファの先頭アドレス－1<br>STOPを入力したときのみ，CYフラグが立つ<br>●使用レジスタ　　　　：すべて |
| 00B7 | BREAKX | CTRL＋STOPキーを押したかどうか調べる<br>●エントリーパラメータ：なし<br>●リターンパラメータ　：<br>押していたらCYフラグが立つ<br>●使用レジスタ　　　　：AF |
| 00BA | ISCNTC | SHIFT＋STOPキーを押したかどうか調べる<br>●エントリーパラメータ：なし<br>●リターンパラメータ　：なし<br>●使用レジスタ　　　　：なし |
| 00BD | CKCNTC | 「ISCNTC」ルーチンと同じ働きをするこのルーチンはBASICで使う<br>●エントリーパラメータ：なし<br>●リターンパラメータ　：なし<br>●使用レジスタ　　　　：なし |
| 00C0 | BEEP | ブザーを鳴らす<br>●エントリーパラメータ：なし<br>●リターンパラメータ　：なし<br>●使用レジスタ　　　　：すべて |
| 00C3 | CLS | 画面をクリアする<br>●エントリーパラメータ：なし<br>●リターンパラメータ　：なし<br>●使用レジスタ　　　　：AF, BC, DE |
| 00C6 | POSIT | 指定した位置にカーソルを移動する<br>●エントリーパラメータ：<br>H←カーソルの水平位置<br>L←カーソルの垂直位置<br>●リターンパラメータ　：なし<br>●使用レジスタ　　　　：AF |
| 00CC | ERAFNK | ファンクションキーの内容の表示を消す<br>●エントリーパラメータ：なし<br>●リターンパラメータ　：なし<br>●使用レジスタ　　　　：すべて |

| アドレス | 名　前 | 機　　能 |
|---|---|---|
| 00CF | DSPFNK | ファンクションキーの内容を表示する<br>●エントリーパラメータ：なし<br>●リターンパラメータ　：なし<br>●使用レジスタ　　　　：すべて |
| 00D2 | TOTEXT | 画面をテキストモードにする<br>●エントリーパラメータ：なし<br>●リターンパラメータ　：なし<br>●使用レジスタ　　　　：すべて |

## ■汎用入出力ポートの入出力ルーチン

| アドレス | 名　前 | 機　　能 |
|---|---|---|
| 00D5 | GTSTCK | ジョイスティックの状態を調べる<br>●エントリーパラメータ：<br>A←調べるジョイスティックの番号<br>●リターンパラメータ　：<br>A←ジョイスティックの向いている方向<br>●使用レジスタ　　　　：すべて |
| 00D8 | GTTRIG | 現在のトリガーボタンの状態を調べる<br>●エントリーパラメータ：<br>A←トリガーボタンの番号<br>●リターンパラメータ　：<br>A＝0ならトリガーボタンは押されていない<br>A＝255ならトリガーボタンが押されている<br>●使用レジスタ　　　　：AF |
| 00DB | GTPAD | 現在のタッチパッドの状態を調べる<br>●エントリーパラメータ：<br>A←タッチパッドの番号<br>●リターンパラメータ　：<br>A←タッチパッドの値<br>●使用レジスタ　　　　：すべて |
| 00DE | GTPDL | 現在のパドルの状態を調べる<br>●エントリーパラメータ：<br>A←パドルの番号<br>●リターンパラメータ　：<br>A←パドルの値<br>●使用レジスタ　　　　：すべて |

## ■カセットの入出力ルーチン

| アドレス | 名　前 | 機　　能 |
|---|---|---|
| 00E1 | TAPION | テープレコーダのモータを動かし，テープのヘッダを読む<br>●エントリーパラメータ：なし<br>●リターンパラメータ　：<br>異常終了したらCYフラグを立てる<br>●使用レジスタ　　　　：すべて |
| 00E4 | TAPIN | テープを読む<br>●エントリーパラメータ：なし<br>●リターンパラメータ　：<br>A←読み込んだ値<br>異常終了したらCYフラグを立てる<br>●使用レジスタ　　　　：すべて |
| 00E7 | TAPIOF | テープからの読み込みを停止する<br>●エントリーパラメータ：なし<br>●リターンパラメータ　：なし<br>●使用レジスタ　　　　：なし |
| 00EA | TAPOON | テープのモータを動かし，ヘッダブロックを書き出す<br>●エントリーパラメータ：<br>A＝0ならショートヘッダ<br>A≠0ならロングヘッダ<br>●リターンパラメータ　：<br>異常終了したらCYフラグを立てる<br>●使用レジスタ　　　　：すべて |
| 00ED | TAPOUT | テープへ値を書き出す<br>●エントリーパラメータ：<br>A←書き出す値<br>●リターンパラメータ　：<br>異常終了したらCYフラグを立てる<br>●使用レジスタ　　　　：すべて |
| 00F0 | TAPOOF | テープへの書き出しを停止する<br>●エントリーパラメータ：なし<br>●リターンパラメータ　：なし<br>●使用レジスタ　　　　：なし |
| 00F3 | STMOTR | テープのモータの動作を指示する<br>●エントリーパラメータ：<br>A＝0　停止<br>A＝1　スタート<br>A＝255　現在と逆の状態にする<br>●リターンパラメータ　：なし<br>●使用レジスタ　　　　：AF |

## ■その他のルーチン

| アドレス | 名　前 | 機　　能 |
|---|---|---|
| 0132 | CHGCAP | CAPランプの状態を変える<br>●エントリーパラメータ：<br>A＝0　ランプを付ける<br>A≠0　ランプを消す<br>●リターンパラメータ　：なし<br>●使用レジスタ　　　　：AF |
| 0135 | CHGSND | サウンドポートの1ビットを変える<br>●エントリーパラメータ：<br>A＝0　ビットをオフ<br>A≠0　ビットをオン<br>●リターンパラメータ　：なし<br>●使用レジスタ　　　　：AF |
| 0141 | SNSMAT | キーボードマトリックスの指定した行の状態を求める<br>●エントリーパラメータ：<br>A←行の番号<br>●リターンパラメータ　：<br>A←キーの状態を表す値<br>キーが押されていたら，対応するビットが0になる<br>●使用レジスタ　　　　：AF |
| 014D | OUTDLP | プリンタに指定した文字を出力する<br>「LPT OUT」ルーチンとの違いは次の点である<br>①TABコードによって空白を印字する<br>②MSX仕様ではないプリンタを使う場合に，〝ひらがな〞〝グラフィック文字〞を変換する<br>③異常終了すると〝device i/o error〞になる<br>●エントリーパラメータ：<br>A←出力するキャラクタコード<br>●リターンパラメータ　：なし<br>●使用レジスタ　　　　：F |
| 0160以降の90バイト | | BIOSのパッチエリア．新しくエントリーを追加するときは，ここに作る |

# 命令変換表

8ビットロード

| × | A | B | C | D | E | H | L | (HL) | (BC) | (DE) | (IX+d) | (IY+d) | (nn) | n | I | R |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| LD A, × | 7F | 78 | 79 | 7A | 7B | 7C | 7D | 7E | 0A | 1A | DD<br>7E<br>d | FD<br>7E<br>d | 3A<br>n<br>n | 3E<br>n | ED<br>57 | ED<br>5F |
| LD B, × | 47 | 40 | 41 | 42 | 43 | 44 | 45 | 46 | | | DD<br>46<br>d | FD<br>46<br>d | | 06<br>n | | |
| LD C, × | 4F | 48 | 49 | 4A | 4B | 4C | 4D | 4E | | | DD<br>4E<br>d | FD<br>4E<br>d | | 0E<br>n | | |
| LD D, × | 57 | 50 | 51 | 52 | 53 | 54 | 55 | 56 | | | DD<br>56<br>d | FD<br>56<br>d | | 16<br>n | | |
| LD E, × | 5F | 58 | 59 | 5A | 5B | 5C | 5D | 5E | | | DD<br>5E<br>d | FD<br>5E<br>d | | 1E<br>n | | |
| LD H, × | 67 | 60 | 61 | 62 | 63 | 64 | 65 | 66 | | | DD<br>66<br>d | FD<br>66<br>d | | 26<br>n | | |
| LD L, × | 6F | 68 | 69 | 6A | 6B | 6C | 6D | 6E | | | DD<br>6E<br>d | FD<br>6E<br>d | | 2E<br>n | | |
| LD (HL), × | 77 | 70 | 71 | 72 | 73 | 74 | 75 | | | | | | | 36<br>n | | |
| LD (BC), × | 02 | | | | | | | | | | | | | | | |
| LD (DE), × | 12 | | | | | | | | | | | | | | | |
| LD (IX+d), × | DD<br>77<br>d | DD<br>70<br>d | DD<br>71<br>d | DD<br>72<br>d | DD<br>73<br>d | DD<br>74<br>d | DD<br>75<br>d | | | | | | | DD<br>36<br>d<br>n | | |
| LD (IY+d), × | FD<br>77<br>d | FD<br>70<br>d | FD<br>71<br>d | FD<br>72<br>d | FD<br>73<br>d | FD<br>74<br>d | FD<br>75<br>d | | | | | | | FD<br>36<br>d<br>n | | |
| LD (nn), × | 32<br>n<br>n | | | | | | | | | | | | | | | |
| LD I, × | ED<br>47 | | | | | | | | | | | | | | | |
| LD R, × | ED<br>4F | | | | | | | | | | | | | | | |

16ビットロード

| × | AF | BC | DE | HL | SP | IX | IY | nn | (nn) |
|---|---|---|---|---|---|---|---|---|---|
| LD AF, × | | | | | | | | | |
| LD BC, × | | | | | | | | 01<br>n<br>n | ED<br>4B<br>n<br>n |
| LD DE, × | | | | | | | | 11<br>n<br>n | ED<br>5B<br>n<br>n |
| LD HL, × | | | | | | | | 21<br>n<br>n | 2A<br>n<br>n |
| LD SP, × | | | | F9 | | DD<br>F9 | FD<br>F9 | 31<br>n<br>n | ED<br>7B<br>n<br>n |
| LD IX, × | | | | | | | | DD<br>21<br>n<br>n | DD<br>2A<br>n<br>n |
| LD IY, × | | | | | | | | FD<br>21<br>n<br>n | FD<br>2A<br>n<br>n |
| LD (nn), × | | ED<br>43<br>n<br>n | ED<br>53<br>n<br>n | 22<br>n<br>n | ED<br>73<br>n<br>n | DD<br>22<br>n<br>n | FD<br>22<br>n<br>n | | |
| PUSH × | F5 | C5 | D5 | E5 | | DD<br>E5 | FD<br>E5 | | |
| POP × | F1 | C1 | D1 | E1 | | DD<br>E1 | FD<br>E1 | | |

ブロック転送

| | |
|---|---|
| LDI | ED<br>A0 |
| LDIR | ED<br>B0 |
| LDD | ED<br>A8 |
| LDDR | ED<br>B8 |

ブロックサーチ

| | |
|---|---|
| CPI | ED<br>A1 |
| CPIR | ED<br>B1 |
| CPD | ED<br>A9 |
| CPDR | ED<br>B9 |

付録に掲載している命令変換表は，Z80のすべての命令を含んでいますので，本書で扱っていない命令もあります．

8ビット算術論理演算

| × | A | B | C | D | E | H | L | (HL) | (IX +d) | (IY +d) | n |
|---|---|---|---|---|---|---|---|---|---|---|---|
| ADD A, × | 87 | 80 | 81 | 82 | 83 | 84 | 85 | 86 | DD 86 d | FD 86 d | C6 n |
| ADC A, × | 8F | 88 | 89 | 8A | 8B | 8C | 8D | 8E | DD 8E d | FD 8E d | CE n |
| SUB × | 97 | 90 | 91 | 92 | 93 | 94 | 95 | 96 | DD 96 d | FD 96 d | D6 n |
| SBC A, × | 9F | 98 | 99 | 9A | 9B | 9C | 9D | 9E | DD 9E d | FD 9E d | DE n |
| AND × | A7 | A0 | A1 | A2 | A3 | A4 | A5 | A6 | DD A6 d | FD A6 d | E6 n |
| XOR × | AF | A8 | A9 | AA | AB | AC | AD | AE | DD AE d | FD AE d | EE n |
| OR × | B7 | B0 | B1 | B2 | B3 | B4 | B5 | B6 | DD B6 d | FD B6 d | F6 n |
| CP × | BF | B8 | B9 | BA | BB | BC | BD | BE | DD BE d | FD BE d | FE n |
| INC × | 3C | 04 | 0C | 14 | 1C | 24 | 2C | 34 | DD 34 d | FD 34 d | |
| DEC × | 3D | 05 | 0D | 15 | 1D | 25 | 2D | 35 | DD 35 d | FD 35 d | |

CPUコントロール

| | |
|---|---|
| NOP | 00 |
| HALT | 76 |
| DI | F3 |
| EI | FB |
| IM 0 | ED 46 |
| IM 1 | ED 56 |
| IM 2 | ED 5E |

16ビット算術演算

| × | BC | DE | HL | SP | IX | IY |
|---|---|---|---|---|---|---|
| ADD HL, × | 09 | 19 | 29 | 39 | | |
| ADD IX, × | DD 09 | DD 19 | | DD 39 | DD 29 | |
| ADD IY, × | FD 09 | FD 19 | | FD 39 | | FD 29 |
| ADC HL, × | ED 4A | ED 5A | ED 6A | ED 7A | | |
| SBC HL, × | ED 42 | ED 52 | ED 62 | ED 72 | | |
| INC × | 03 | 13 | 23 | 33 | DD 23 | FD 23 |
| DEC × | 0B | 1B | 2B | 3B | DD 2B | FD 2B |

エクスチェンジ

| | |
|---|---|
| EX AF, AF' | 08 |
| EX DE, HL | EB |
| EX (SP), HL | E3 |
| EX (SP), IX | DD E3 |
| EX (SP), IY | FD E3 |
| EXX | D9 |

アキュムレータ操作

| | |
|---|---|
| DAA | 27 |
| CPL | 2F |
| NEG | ED 44 |
| CCF | 3F |
| SCF | 37 |

## ローテート，シフト

| ×| A | B | C | D | E | H | L | (HL) | (IX +d) | (IY +d) |
|---|---|---|---|---|---|---|---|---|---|---|
| RLC × | CB<br>07 | CB<br>00 | CB<br>01 | CB<br>02 | CB<br>03 | CB<br>04 | CB<br>05 | CB<br>06 | DD<br>CB<br>d<br>06 | FD<br>CB<br>d<br>06 |
| RRC × | CB<br>0F | CB<br>08 | CB<br>09 | CB<br>0A | CB<br>0B | CB<br>0C | CB<br>0D | CB<br>0E | DD<br>CB<br>d<br>0E | FD<br>CB<br>d<br>0E |
| RL × | CB<br>17 | CB<br>10 | CB<br>11 | CB<br>12 | CB<br>13 | CB<br>14 | CB<br>15 | CB<br>16 | DD<br>CB<br>d<br>16 | FD<br>CB<br>d<br>16 |
| RR × | CB<br>1F | CB<br>18 | CB<br>19 | CB<br>1A | CB<br>1B | CB<br>1C | CB<br>1D | CB<br>1E | DD<br>CB<br>d<br>1E | FD<br>CB<br>d<br>1E |
| SLA × | CB<br>27 | CB<br>20 | CB<br>21 | CB<br>22 | CB<br>23 | CB<br>24 | CB<br>25 | CB<br>26 | DD<br>CB<br>d<br>26 | FD<br>CB<br>d<br>26 |
| SRA × | CB<br>2F | CB<br>28 | CB<br>29 | CB<br>2A | CB<br>2B | CB<br>2C | CB<br>2D | CB<br>2E | DD<br>CB<br>d<br>2E | FD<br>CB<br>d<br>2E |
| SRL × | CB<br>3F | CB<br>38 | CB<br>39 | CB<br>3A | CB<br>3B | CB<br>3C | CB<br>3D | CB<br>3E | DD<br>CB<br>d<br>3E | FD<br>CB<br>d<br>3E |
| RLD | | | | | | | | ED<br>6F | | |
| RRD | | | | | | | | ED<br>67 | | |

| | A |
|---|---|
| RLCA | 07 |
| RRCA | 0F |
| RLA | 17 |
| RRA | 1F |

## ジャンプ，コール，リターン

| × | 無条件 | キャリー<br>C | ノンキャリー<br>NC | ゼロ<br>Z | ノンゼロ<br>NZ | パリティ偶数<br>PE | パリティ奇数<br>PO | 負<br>M | 正<br>P | カウント |
|---|---|---|---|---|---|---|---|---|---|---|
| JP ×, nn | C3<br>n<br>n | DA<br>n<br>n | D2<br>n<br>n | CA<br>n<br>n | C2<br>n<br>n | EA<br>n<br>n | E2<br>n<br>n | FA<br>n<br>n | F2<br>n<br>n | |
| JR ×, e | 18<br>e−2 | 38<br>e−2 | 30<br>e−2 | 28<br>e−2 | 20<br>e−2 | | | | | |
| JP (HL) | E9 | | | | | | | | | |
| JP (IX) | DD<br>E9 | | | | | | | | | |
| JP (IY) | FD<br>E9 | | | | | | | | | |
| CALL ×, nn | CD<br>n<br>n | DC<br>n<br>n | D4<br>n<br>n | CC<br>n<br>n | C4<br>n<br>n | EC<br>n<br>n | E4<br>n<br>n | FC<br>n<br>n | F4<br>n<br>n | |
| DJNZ e | | | | | | | | | | 10<br>e−2 |
| RET × | C9 | D8 | D0 | C8 | C0 | E8 | E0 | F8 | F0 | |
| RETI | ED<br>4D | | | | | | | | | |
| RETN | ED<br>45 | | | | | | | | | |

## リスタート

| | |
|---|---|
| RST 00H | C7 |
| RST 08H | CF |
| RST 10H | D7 |
| RST 18H | DF |
| RST 20H | E7 |
| RST 28H | EF |
| RST 30H | F7 |
| RST 38H | FF |

ビット操作

| × | A | B | C | D | E | H | L | (HL) | (IX +d) | (IY +d) |
|---|---|---|---|---|---|---|---|---|---|---|
| BIT 0, × | CB 47 | CB 40 | CB 41 | CB 42 | CB 43 | CB 44 | CB 45 | CB 46 | DD CB d 46 | FD CB d 46 |
| BIT 1, × | CB 4F | CB 48 | CB 49 | CB 4A | CB 4B | CB 4C | CB 4D | CB 4E | DD CB d 4E | FD CB d 4E |
| BIT 2, × | CB 57 | CB 50 | CB 51 | CB 52 | CB 53 | CB 54 | CB 55 | CB 56 | DD CB d 56 | FD CB d 56 |
| BIT 3, × | CB 5F | CB 58 | CB 59 | CB 5A | CB 5B | CB 5C | CB 5D | CB 5E | DD CB d 5E | FD CB d 5E |
| BIT 4, × | CB 67 | CB 60 | CB 61 | CB 62 | CB 63 | CB 64 | CB 65 | CB 66 | DD CB d 66 | FD CB d 66 |
| BIT 5, × | CB 6F | CB 68 | CB 69 | CB 6A | CB 6B | CB 6C | CB 6D | CB 6E | DD CB d 6E | FD CB d 6E |
| BIT 6, × | CB 77 | CB 70 | CB 71 | CB 72 | CB 73 | CB 74 | CB 75 | CB 76 | DD CB d 76 | FD CB d 76 |
| BIT 7, × | CB 7F | CB 78 | CB 79 | CB 7A | CB 7B | CB 7C | CB 7D | CB 7E | DD CB d 7E | FD CB d 7E |
| RES 0, × | CB 87 | CB 80 | CB 81 | CB 82 | CB 83 | CB 84 | CB 85 | CB 86 | DD CB d 86 | FD CB d 86 |
| RES 1, × | CB 8F | CB 88 | CB 89 | CB 8A | CB 8B | CB 8C | CB 8D | CB 8E | DD CB d 8E | FD CB d 8E |
| RES 2, × | CB 97 | CB 90 | CB 91 | CB 92 | CB 93 | CB 94 | CB 95 | CB 96 | DD CB d 96 | FD CB d 96 |
| RES 3, × | CB 9F | CB 98 | CB 99 | CB 9A | CB 9B | CB 9C | CB 9D | CB 9E | DD CB d 9E | FD CB d 9E |
| RES 4, × | CB A7 | CB A0 | CB A1 | CB A2 | CB A3 | CB A4 | CB A5 | CB A6 | DD CB d A6 | FD CB d A6 |
| RES 5, × | CB AF | CB A8 | CB A9 | CB AA | CB AB | CB AC | CB AD | CB AE | DD CB d AE | FD CB d AE |
| RES 6, × | CB B7 | CB B0 | CB B1 | CB B2 | CB B3 | CB B4 | CB B5 | CB B6 | DD CB d B6 | FD CB d B6 |
| RES 7, × | CB BF | CB B8 | CB B9 | CB BA | CB BB | CB BC | CB BD | CB BE | DD CB d BE | FD CB d BE |
| SET 0, × | CB C7 | CB C0 | CB C1 | CB C2 | CB C3 | CB C4 | CB C5 | CB C6 | DD CB d C6 | FD CB d C6 |
| SET 1, × | CB CF | CB C8 | CB C9 | CB CA | CB CB | CB CC | CB CD | CB CE | DD CB d CE | FD CB d CE |
| SET 2, × | CB D7 | CB D0 | CB D1 | CB D2 | CB D3 | CB D4 | CB D5 | CB D6 | DD CB d D6 | FD CB d D6 |
| SET 3, × | CB DF | CB D8 | CB D9 | CB DA | CB DB | CB DC | CB DD | CB DE | DD CB d DE | FD CB d DE |
| SET 4, × | CB E7 | CB E0 | CB E1 | CB E2 | CB E3 | CB E4 | CB E5 | CB E6 | DD CB d E6 | FD CB d E6 |
| SET 5, × | CB EF | CB E8 | CB E9 | CB EA | CB EB | CB EC | CB ED | CB EE | DD CB d EE | FD CB d EE |
| SET 6, × | CB F7 | CB F0 | CB F1 | CB F2 | CB F3 | CB F4 | CB F5 | CB F6 | DD CB d F6 | FD CB d F6 |
| SET 7, × | CB FF | CB F8 | CB F9 | CB FA | CB FB | CB FC | CB FD | CB FE | DD CB d FE | FD CB d FE |

入　力

| | |
|---|---|
| IN A, n | DB n |
| IN A, (C) | ED 78 |
| IN B, (C) | ED 40 |
| IN C, (C) | ED 48 |
| IN D, (C) | ED 50 |
| IN E, (C) | ED 58 |
| IN H, (C) | ED 60 |
| IN L, (C) | ED 68 |
| INI | ED A2 |
| INIR | ED B2 |
| IND | ED AA |
| INDR | ED BA |

出　力

| | |
|---|---|
| OUT n, A | D3 n |
| OUT (C), A | ED 79 |
| OUT (C), B | ED 41 |
| OUT (C), C | ED 49 |
| OUT (C), D | ED 51 |
| OUT (C), E | ED 59 |
| OUT (C), H | ED 61 |
| OUT (C), L | ED 69 |
| OUTI | ED A3 |
| OTIR | ED B3 |
| OUTD | ED AB |
| OTDR | ED BB |

# フラグへの影響

| 命令 | S | Z | | H | | P/V | N | CY | コメント |
|---|---|---|---|---|---|---|---|---|---|
| ADD A, s ; ADC A, s | ↕ | ↕ | × | ↕ | × | V | 0 | ↕ | 8ビット足し算命令 |
| SUB s ; SBC A, s ; CP s ; NEG | ↕ | ↕ | × | ↕ | × | V | 1 | ↕ | 8ビット引き算・比較・NEG命令 |
| AND s | ↕ | ↕ | × | 1 | × | P | 0 | 0 | 論理演算命令 |
| OR s ; XOR s | ↕ | ↕ | × | 0 | × | P | 0 | 0 | |
| INC s | ↕ | ↕ | × | ↕ | × | V | 0 | • | 8ビット増減命令 |
| DEC s | ↕ | ↕ | × | ↕ | × | V | 1 | • | |
| ADD dd, ss | • | • | × | × | × | • | 0 | ↕ | 16ビット足し算命令 |
| ADC HL, ss | ↕ | ↕ | × | × | × | V | 0 | ↕ | 16ビット足し算命令（キャリー含む） |
| SBC HL, ss | ↕ | ↕ | × | × | × | V | 1 | ↕ | 16ビット引き算命令（キャリー含む） |
| RLA ; RLCA ; RRA ; RRCA | • | • | × | 0 | × | • | 0 | ↕ | ローテイト・アキュムレータ命令 |
| RLs ; RLC s ; RR s ; RRC s<br>SLA s ; SRA s ; SRL s | ↕ | ↕ | × | 0 | × | P | 0 | ↕ | ローテイト／シフト命令 |
| RLD ; RRD | ↕ | ↕ | × | 0 | × | P | 0 | • | ローテイト・デジット命令 |
| DAA | ↕ | ↕ | × | ↕ | × | P | • | ↕ | 10進補正演算命令 |
| CPL | • | • | × | 1 | × | • | 1 | • | アキュムレータ反転命令 |
| SCF | • | • | × | 0 | × | • | 0 | 1 | キャリーセット命令 |
| CCF | • | • | × | × | × | • | 0 | ↕ | キャリー反転命令 |
| IN r, (C) | ↕ | ↕ | × | ↕ | × | P | 0 | • | 入力ポート指定命令 |
| INI ; IND ; OUTI ; OUTD | × | ↕ | × | × | × | × | 1 | • | ブロックI/O命令 |
| INIR ; INDR ; OTIR ; OTDR | × | 1 | × | × | × | × | 1 | • | B−1≠0ならばZ=0, その他はZ=1 |
| LDI ; LDD | • | • | × | 0 | × | ↕ | 0 | • | ブロック転送命令 |
| LDIR ; LDDR | • | • | × | 0 | × | 0 | 0 | • | BC−1≠0ならばP/V=1, その他はP/V=0 |
| CPI ; CPIR<br>CPD ; CPDR | ↕ | ↕ | × | ↕ | × | ↕ | 1 | • | ブロックサーチ命令<br>A＝(HL)ならばZ=1, その他はZ=0<br>BC−1≠0ならばP/V=1, その他はP/V=0 |
| LD A, I ; LD A, R | ↕ | ↕ | × | 0 | × | IFF | 0 | • | IFFの内容をP/Vにコピー |
| BIT b, s | × | ↕ | × | 1 | × | × | 0 | • | sのビットbの内容をZにコピー |

s ：8ビットのレジスタおよびメモリ
ss：16ビットのレジスタおよびメモリ
r ：8ビットのレジスタ

ここに掲載されていない命令は，命令実行後にフラグに影響を与えない．

# キャラクタコード表

| 上位<br>下位 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F | 4 | 5 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | コントロールコード | | | 0 | @ | P | ` | p | ♠ | | | ー | タ | ミ | た | み | | π |
| 1 | | | ! | 1 | A | Q | a | q | ♥ | あ | 。 | ア | チ | ム | ち | む | 月 | ┴ |
| 2 | | | " | 2 | B | R | b | r | ♣ | い | 「 | イ | ツ | メ | つ | め | 火 | ┬ |
| 3 | | | # | 3 | C | S | c | s | ♦ | う | 」 | ウ | テ | モ | て | も | 水 | ┤ |
| 4 | | | $ | 4 | D | T | d | t | ○ | え | 、 | エ | ト | ヤ | と | や | 木 | ├ |
| 5 | | | % | 5 | E | U | e | u | ● | お | ・ | オ | ナ | ユ | な | ゆ | 金 | ┼ |
| 6 | | | & | 6 | F | V | f | v | を | か | ヲ | カ | ニ | ヨ | に | よ | 土 | │ |
| 7 | | | ' | 7 | G | W | g | w | ぁ | き | ァ | キ | ヌ | ラ | ぬ | ら | 日 | ─ |
| 8 | | | ( | 8 | H | X | h | x | ぃ | く | ィ | ク | ネ | リ | ね | り | 年 | ┌ |
| 9 | | | ) | 9 | I | Y | i | y | ぅ | け | ゥ | ケ | ノ | ル | の | る | 円 | ┐ |
| A | | | * | : | J | Z | j | z | ぇ | こ | ェ | コ | ハ | レ | は | れ | 時 | └ |
| B | | | + | ; | K | [ | k | ¦ | ぉ | さ | ォ | サ | ヒ | ロ | ひ | ろ | 分 | ┘ |
| C | | | , | < | L | ¥ | l | ¦ | ゃ | し | ャ | シ | フ | ワ | ふ | わ | 秒 | ╳ |
| D | | | − | = | M | ] | m | ¦ | ゅ | す | ュ | ス | ヘ | ン | へ | ん | 百 | 大 |
| E | | | . | > | N | ^ | n | ~ | ょ | せ | ョ | セ | ホ | ゛ | ほ | | 千 | 中 |
| F | | | ／ | ? | O | _ | o | | っ | そ | ッ | ソ | マ | ゜ | ま | ■ | 万 | 小 |

CHR$(&H41)＝"A"になる．
グラフィックキャラクタコードは最初にCHR$(1)をつける．
CHR$(1)＋CHR$(&H41)＝"月"

# 索　引

**参 考 文 献**

1)「MSXテクニカルデータブック 1」アスキー発行
2)「はじめて語むマシン語」村瀬康治著　アスキー発行
3)「PC-6001mkII マシン語入門」市川他共著　アスキー発行
4)「PC-8801 マシン語入門」藤井他共著　アスキー発行
5)「MSX早わかり辞典」朝日新聞社発行
6)「MSXポケットバンク 4　マイコン・サウンドパック」工藤賢司著　アスキー発行
7)「MSXポケットバンク 9　グラフィックス㊙伝」安田吾郎著　アスキー発行
8)「MSXポケットバンク 11　とにかく速いマシン語ゲーム集」
ポケットバンク編集部著　アスキー発行

## MSXマシン語入門講座

1984年10月25日　初版発行
1987年9月11日　第1版第8刷発行
定価1,600円

著　者　湯浅　敬
発行者　塚本慶一郎
発行所　株式会社アスキー
〒107 東京都港区南青山6-11-1スリーエフ南青山ビル
振　替　東京4-161144
TEL (03)486-7111(大代表)
情報TEL (03)498-0299(ダイヤルイン)
出版営業部TEL (03)486-1977(ダイヤルイン)



編集担当　土屋信明
表紙担当　郷　啓子
印刷　壮光舎印刷株式会社

ISBN4-87148-761-X C3055 ¥1600E



ISBN4-87148-761-X C3055 ¥1600E 定価1,600円